5-6-2023

AN INVESTIGATION ON THE RESILIENCE OF LONG SHORT-TERM MEMORY DEEP NEURAL NETWORKS

Christopher Vasquez
Louisiana State University and Agricultural and Mechanical College

AN INVESTIGATION ON THE RESILIENCE OF LONG SHORT-TERM MEMORY DEEP NEURAL NETWORKS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Christopher Paul Vasquez
B.S.Comp.E., Louisiana State University, 2022
B.S.Comp.S., Louisiana State University, 2022
August 2023

Acknowledgments

Firstly, I would like to thank Dr. Lu Peng and Dr. Travis LeCompte for their guidance and assistance throughout the duration of this work. Additionally, I would like to thank Dr. Jerry Trahan and Dr. Kidong Park for making multiple accommodations for my research during my graduate years at Louisiana State University (LSU). Finally, I want to thank Dr. Genevieve Palardy, Dr. Nash Mahmoud, Dr. Andrew Webb, Dr. Bin Li, Dr. Ramachandran Vaidyanathan, and Dr. Steven Brandt for providing various opportunities and additional guidance throughout my time at LSU.

Contents

List of Tables

List of Figures

Abstract

In a world of continuously advancing technology, the reliance on these technologies continues to increase. Recently, transformer networks [22] have been implemented through various projects such as ChatGPT. These networks are extremely computationally demanding and require cutting-edge hardware to explore. However, with the growing increase and popularity of these neural networks, a question of reliability and resilience comes about, especially as the dependency and research on these networks grow. Given the computational demand of transformer networks, we investigate the resilience of the weights and biases of the predecessor of these networks, i.e. the Long Short-Term (LSTM) neural network, through four implementations of the original LSTM network. Based on the observations made through fault injection of these networks, we propose an effective means of fault mitigation through Hamming encoding of selected weights and biases in a given network and lay the groundwork for similar mitigation methods with transformers.

Chapter 1. Introduction

Wanting to explore the resilience of LSTM networks, we developed four different models with various architectures. All developed models were constructed for the task of binary sentiment analysis. By introducing various numbers of errors under different conditions, we were able to determine areas of susceptibility that lead to decreases in each model's accuracy with respect to the weights and biases. Based on these findings, we developed a means of mitigating the errors by encoding the most susceptible bits of the weights and biases. In an effort to reduce computation costs, we also only selected specific sets of these weights and biases to encode based on their susceptibility. Implementing our method into each network, we compared and proved the effectiveness of our fault mitigation method for improving the resilience of LSTM networks.

Chapter 2. Theoretical Framework

2.1. Background

2.1.1. Neural Networks

As research into artificial intelligence (AI) continues to advance, various systems have come to rely on machine learning in various different mechanical, electrical, and computerized systems. As the name implies, machine learning can be described as the ability of a machine to adapt to environments and situations without the need for explicit instructions or programming. Furthermore, a subset of machine learning is deep learning which more closely aligns with AI. Despite the various high-level definitions of deep learning, [6] identifies two main key aspects amongst them: (1) models consisting of multiple layers or stages of nonlinear information processing; and (2) methods for supervised or unsupervised learning of feature representation at successively higher, more abstract layers. The first aspect describes the typical structure of deep learning structures known as neural networks. As noted by the second point, these networks are tasked with learning sets of information, as known as features, by processing them through each layer of the neural network.

Expanding upon the architecture of neural networks, each typically consists of neurons organized into layers. A neuron can be defined as an individual, minimal mathematical function. Each neuron usually consists of a set of weighted inputs to be summed before being passed to a non-linear activation function. This activation function serves as a threshold function to determine if a neuron's output should be active or not. Additionally, individual layers of a neural network can possess a bias to influence the weighted inputs.

Figure 2.1. shows an example of a typical neuron.

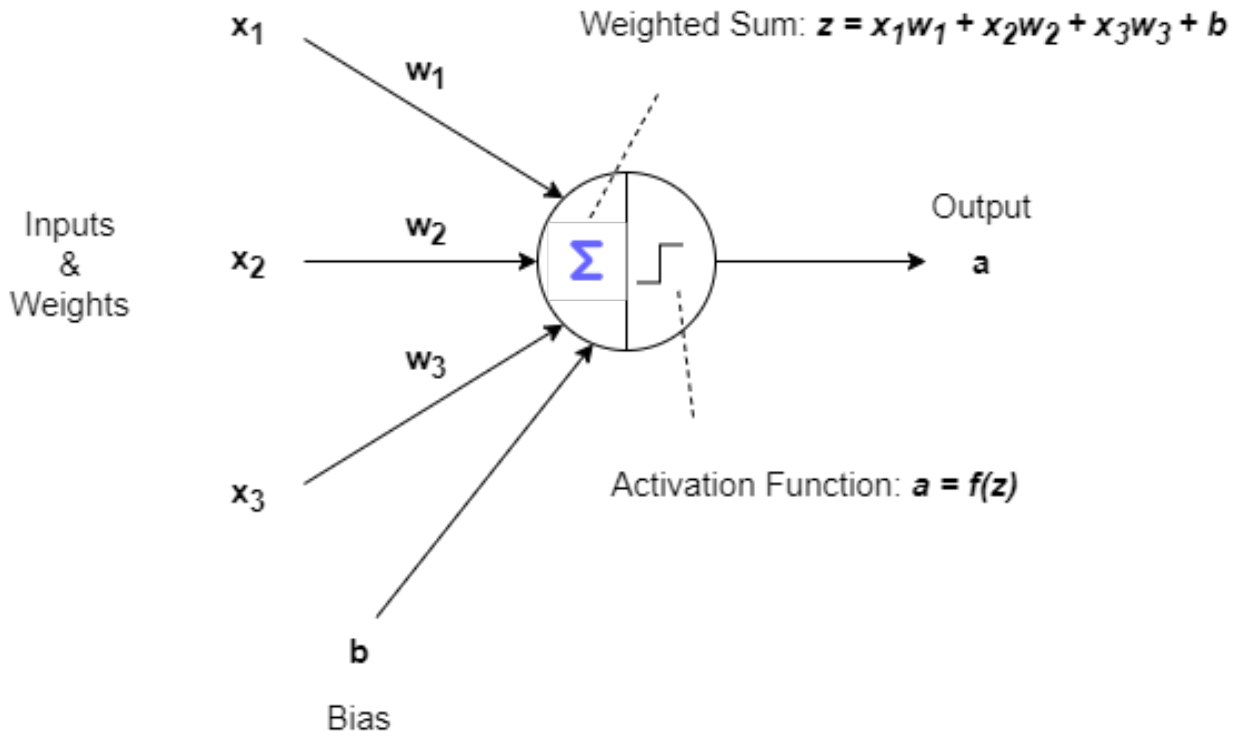


Figure 2.1. Depiction of Neuron with 3 Inputs and Weights

With regards to the architecture of a neural network, the first layer is called the *input layer* and contains a set of input neurons. The last layer of a neural network is called the *output layer* and consists of any given number of output neurons depending on the architecture of the network. Neural networks can have any amount of layers between the input and output layers, all of which are known as *hidden layers*. To connect each layer, the neurons within each are connected to the previous layer's neurons. If all the neurons of a given layer are connected to all of the neurons of the previous layer, the layer in question is considered *fully connected*. If a network consists of only fully-connected layers, it is called a multi-layer perceptron; this type of network can be seen in Figure 2.2.. The network depicted in the figure can also be described as *feedforward* since data flows from the

input layers, through the hidden layers, and out the output layer directly [18].

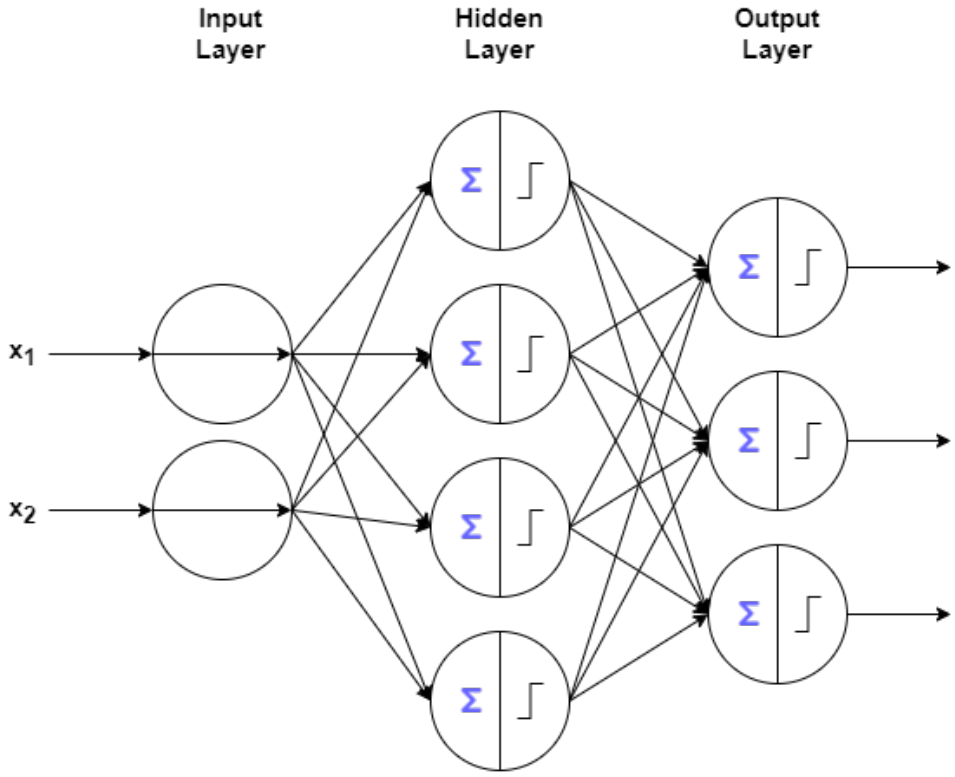


Figure 2.2. Depiction of Perceptron with 2 Inputs and 3 Outputs

To calculate the weights and biases of a neural network, it must undergo a process known as *training*. When a neural network is first created, its weights and biases are initialized randomly. During the training process, the neural network iteratively adjusts these random values through a process known as *back-propagation* [20]. This process minimizes the error between predictions and the actual target values using a combination of forward and backward passes through the network. Once the weights have been adjusted, the model can be deployed and undergo *inference*, where prediction can be made on relevant data or samples.

2.1.2. Recurrent Neural Network (RNN)

RNNs are one of the most common types of deep neural networks due to their versatility in solving problems such as speech recognition, language modeling, image captioning, and many others. Unlike feed-forward neural networks, this type of network contains feedback paths, allowing for information to persist. As seen in Figure 2.3., the output of the hidden layers is cycled back to the same hidden layers for the next iteration, allowing previous information to be utilized.

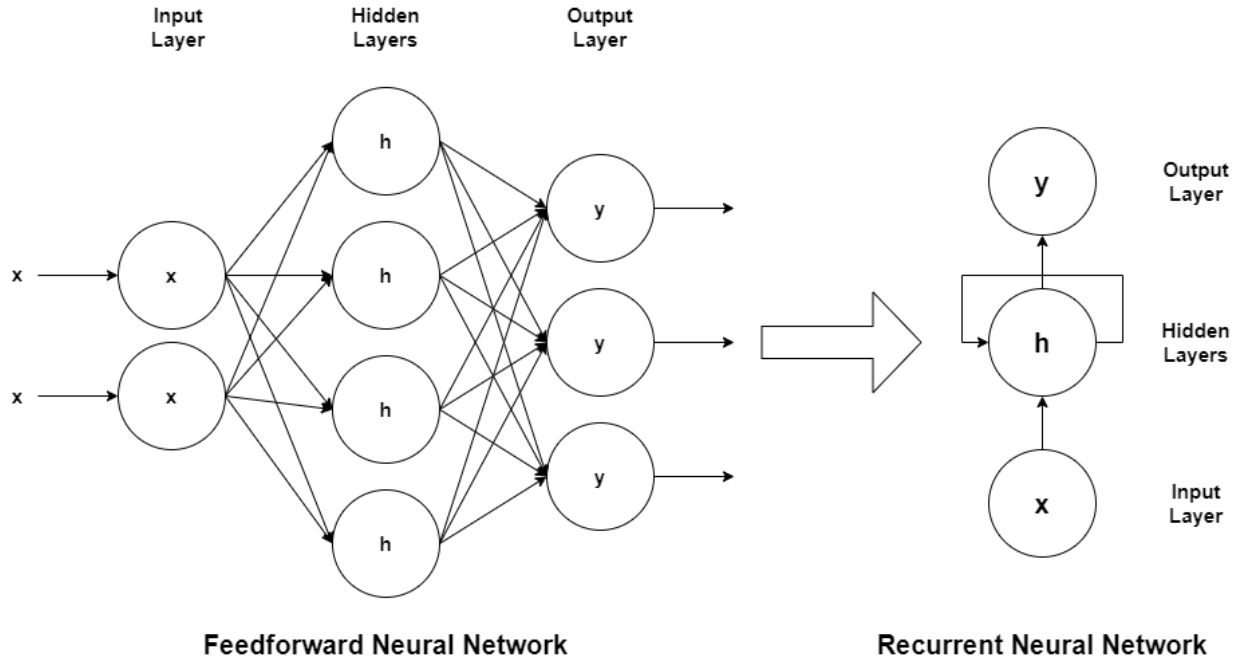


Figure 2.3. Depiction of Feed-forward Neural Network Conversion to Recurrent Neural Network

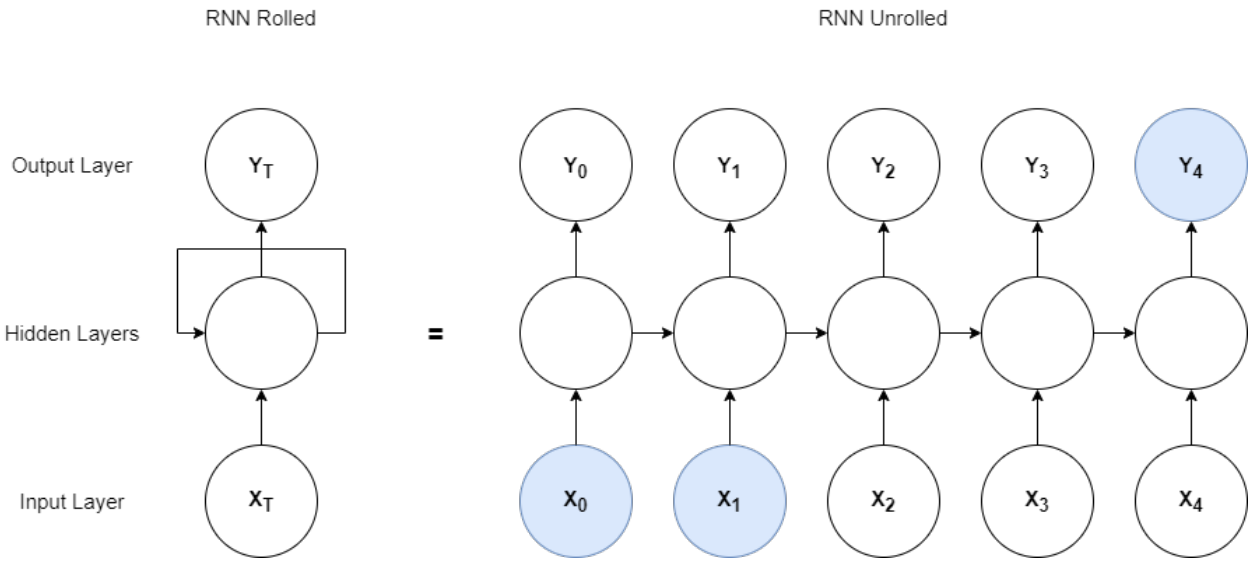
One of the main appeals of RNNs is their ability to connect previous information to a present task. For instance, consider a language model for predicting the next word based on previous ones. If we are predicting the last word in "wood comes from *trees*," no further context is needed, as the gap between the relevant information and the predicted word is small, as depicted in Figure 2.4.a. In such cases, RNNs can utilize past informa-

tion due to their usage of persistent information. However, certain cases may require more context. Consider the same language model trying to predict the last word in "I lived in England... I speak fluent *English*." The most recent information–i.e. "I speak fluent..."– suggests that the next word is the name of a language. However, to determine which language, the model requires the previous context. Given this circumstance, the gap between relevant information and the predicted word can grow to be very large, as seen in Figure Figure 2.4.b. Furthermore, as the gap grows, RNNs tend to become less efficient in learning to connect information.

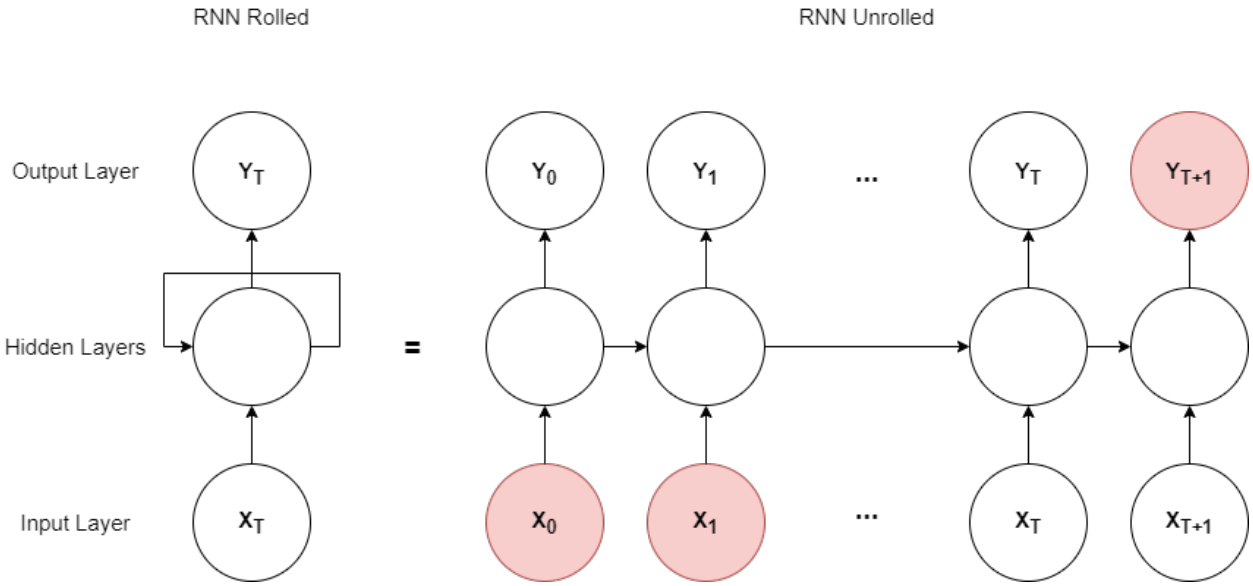
2.1.3. Long Short-Term Memory (LSTM)

As a solution to the limited lifespan of RNNs' memory persistence, Long Short Term Memory networks were developed, being a class of RNN capable of long-term dependencies. Introduced by Hochreiter & Schmidhuber in [12], these networks maintain the versatility of RNNs while also avoiding the long-term dependency problem, explaining their wide use. Like any RNN, LSTM networks form a chain of repeating modules. However, unlike standard RNNs which typically use one neural network layer for the feedback, LSTMs utilize four specific neural network layers to allow their long short-term memory behavior, as seen in Figure 2.5. [19]. Each instance of these four layers, i.e. the LSTM network, is called a cell [12]

With regard to the diagram, each line in the figure carries a vector from the output of one neuron to the input of another. Red circles represent pointwise operations, such as vector addition and vector multiplication, while the yellow boxes represent the learned neural network layers. Merging lines denote concatenation while a forking line denotes a



(a) RNN with Small Gap Between Relevant Information



(b) RNN with Large Gap Between Relevant Information

Figure 2.4. Illustrations of RNN Gap Differences

vector being copied to different locations.

The core of the LSTMs' functionality lies in the horizontal path, as seen in Figure 2.6.b that runs through the top of the cells and serves as a conveyor belt for information. This "conveyor belt" of information can also be called the cell state. LSTMs are

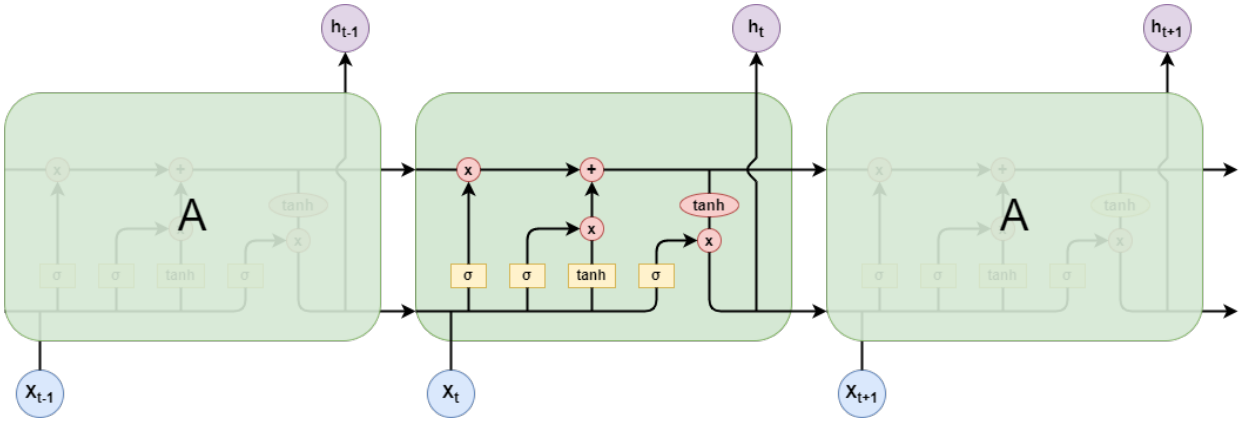
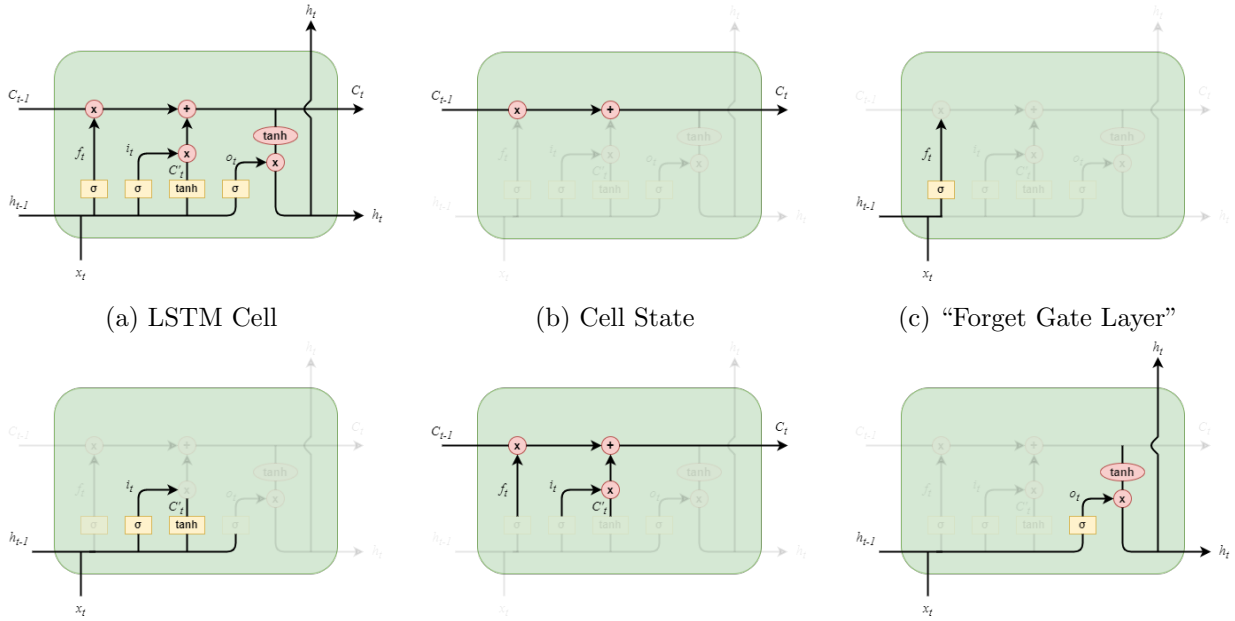


Figure 2.5. LSTM Cell with Four Interacting Neural Network Layers for Long-Term Data Persistence



(a) LSTM Cell (b) Cell State (c) "Forget Gate Layer"

(d) "Input Gate Layer" (e) Pointwise Operations (f) "Output Filter Layer"

Figure 2.6. LSTM Module Process for Long-Term Persistent Data

able to add and remove information from the cell state via structures known as gates [12]. Gates consist of the sigmoid neural network layer and a pointwise multiplication operation. These structures are able to control the cell state by determining whether or not to let information pass through to the cell state.

Given that an LSTM contains four distinct neural network layers, its behavior can

be divided into four main parts. As seen in Figure 2.6.c, the first layer of the LSTM determines what information is going to be filtered out of the cell state. Due to this behavior, this layer is referred to as the "forget gate." Specifically, the sigmoid layer examines the values of h_{t-1} and x_t and outputs a number between 0 and 1 for every number in cell state C_{t-1}. A value of 1 denotes that the information should be kept while a 0 means the information should be forgotten. The output f_t of the "forget gate" can be mathematically represented by equation 2.1. With further simplification, we can see the weights and bias variables that the neural network uses to influence f_t during training; these variables include W_f, U_f, and b_f. This simplification will be performed for all subsequent equations presented within this section.

$$\begin{aligned} f_t &= \sigma(W_f * [h_{t-1}, x_t] + b_f) \\ &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \end{aligned} \tag{2.1}$$

The next process of the neural network decides which new information will be stored in the cell state, as depicted in Figure 2.6.d. This process is divided into two steps. The sigmoid layer will first decide which values to update in the cell state, i_t, giving this layer the name the "input gate layer." Then, the tanh layer will create a vector of potential values, C'_t, to be added to the state. Equations 2.2 and 2.3 more formally describe the outputs of this process.

$$\begin{aligned} i_t &= \sigma(W_i * [h_{t-1}, x_t] + b_i) \\ &= \sigma(W_i x_t + U_i h_t + b_i) \end{aligned} \tag{2.2}$$

$$\begin{aligned} C'_t &= \tanh(W_c * [h_{t-1}, x_t] + b_C) \\ &= \tanh(W_c x_t + U_f h_{t-1} + b_C) \end{aligned} \tag{2.3}$$

With the previous outputs f_t, i_t, and C'_t, we can update the old cell state C_{t-1} to the new cell state C_t. As seen in Figure 2.6.e, we perform a pointwise multiplication on the old state C_{t-1} with the "forget gate layer output" f_t. This operation will allow the state to forget the values previously decided. The LSTM then adds $i_t * C'_t$–the new potential values C'_t scaled by the amount i_t to update each value–to the previously computed product, producing the new state C_t. This process can also be described by equation 2.4.

$$\begin{aligned} C_t &= \sigma(f_t * C_{t-1} + i_t * C'_t) \\ &= \sigma(f_t \circ C_{t-1} + i_t \circ C'_t) \end{aligned} \tag{2.4}$$

As a final step, the LSTM determines the information h_t to output by filtering the new state C_t. As seen in Figure 2.6.f, the sigmoid layer decides the output state o_t, i.e. which values are going to be output. Additionally, the cell state is processed through tanh, bounding the values between -1 and 1. Then, we multiply the tanh output with the sigmoid output to only output the values we previously determined, i.e. the hidden state h_t. This final step can be mathematically described by equations 2.5 and 2.6.

$$\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \end{aligned} \tag{2.5}$$

$$\begin{aligned} h_t &= o_t * \tanh(C_t) \\ &= o_t \circ \tanh(C_t) \end{aligned} \tag{2.6}$$

There are many other variations of LSTMs that will not be covered, as they are beyond the scope of this work.

2.2. Fault Injection

2.2.1. Fault Tolerance and Safety

From self-driving cars to usage in nuclear power plants, the gravity of neural networks' consistent accuracy becomes increasingly important for ensuring the safety of others. Given that neural networks are employed in a variety of environments, they may undergo various conditions, such as high temperatures/altitude zones/vehicles, which make them more susceptible to errors occurring. These types of soft errors can potentially lead to application failures that violate the safety and reliability specifications of a neural network. Although the performance of these networks has been extensively studied, the reliability aspects of their usage are not as well understood [16].

The fault-tolerance of a system is defined as the ability of a system to carry out its task after encountering hardware or software defects [13]. In this study, we focus on hardware faults that can arise from external factors, affecting the hardware of a system. Two major categories of these faults are permanent and transient faults [10]. The former fault typically results from a physical flaw within the hardware such as a short circuit. However, the more common type of fault, i.e. transient faults, can result from high-energy particles, striking electronic devices. An example of a transient fault, also known as a soft error [24], is a Single Event Upset (SEU). These SEUs occur when a bit in a storage ele-

ment becomes flipped to another state. Many studies have been conducted to research and improve the resilience of hardware and software due to these anomalies [2][8][9][14][15].

2.2.2. Single-Precision Floating-Point Number Format

One of the ways numbers can be represented in computer systems is through floating-point format. In this format, numbers consist of a *sign*, *mantissa*, *exponent*, and *base*. However, in the context of computer systems, the *base* can be assumed to be 2. Therefore, a sign, mantissa, and exponent are the only components needed to represent floating-point numbers in computer systems. The technical standard for floating point arithmetic and representation is known as IEEE-754. Specifically, we will be working exclusively with the single-precision floating-point format shown in Figure 2.7.. The sign, mantissa, and exponent have 1, 23, and 8 bits, respectively. This format will play an important role, as we will be randomly flipping the bits of this format.

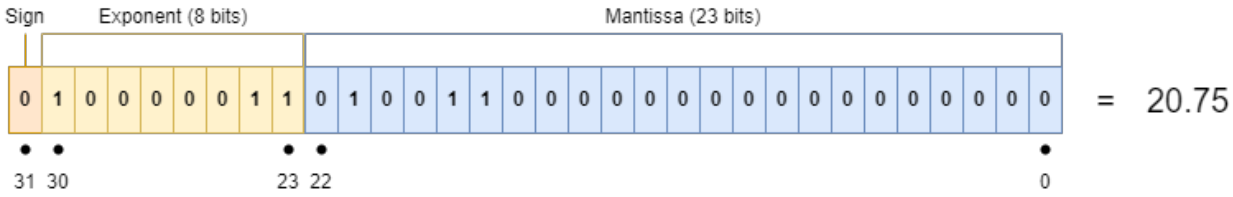


Figure 2.7. IEEE-754 Single-Precision Floating-Point Format

2.3. Related Work

2.3.1. Single Event Upsets (SEUs) and Deep Neural Networks

Within [23], various observations and remedies regarding the fault tolerance of neural networks were made. Specifically, the work explores the effects of faults on deep neural networks, using ResNet56 for all experiments. Many of their conclusions align with the same finding of this work, such as the location of most significant susceptibility being

the exponent segment of a single-precision floating-point format neural network weight. In addition to multiple findings, they propose triple modular redundancy (TMR) and error-correcting code as potential solutions to combat transient faults. Specifically, the paper suggests hamming code. However, utilizing such code would require more bits. As seen in our work, this may potentially be more harmful than helpful. However, based on the counterpart suggestion of the paper, we developed a type of fault-mitigation and weight-recovery method for our neural network.

2.3.2. The Impact of Faults on DNNs: A Case Study

Similar to our work, Malekzadeh el al (2021) explored the fault injection (FI) resilience of LeNet_5–a convolutional neural network–with methods similar to ours in [18]. Specifically, this author tested the effects of faults within each layer and developed architectural methods to mitigate the faults within the network. A potential solution of the work was the utilization of the sigmoid activation function to limit the range of output values. Given that our study focuses on LSTMs, these networks already natively make use of this activation function. Therefore, an alternative method of fault protection can still be explored. Additionally, wanting to expand on this work, we have chosen to increase the number of injections performed and similar solutions to potentially mitigate fault impact on LSTM networks. With this work in mind, we are hoping to observe some similarities between the two to generalize the effects of FI on neural networks as a whole.

2.3.3. Ranger: A Low-cost Fault Corrector

In an effort to reduce the impact of faults in output layers of neural networks, [3] proposes a range restriction method applied to all outputs of a layer. This method

of mitigation is low-cost and has been tested with various convolutional neural networks (CNNs)–namely AlexNet and NVIDIA Dave among others. By bounding outputs to a specific range, Chen et al (2021) avoids the potential "explosion" of output values when SEUs occur in the exponent bit field. Such a method has proven to work well with layer outputs but has not been implemented with the weights or biases of networks.

Chapter 3. Design & Implementation

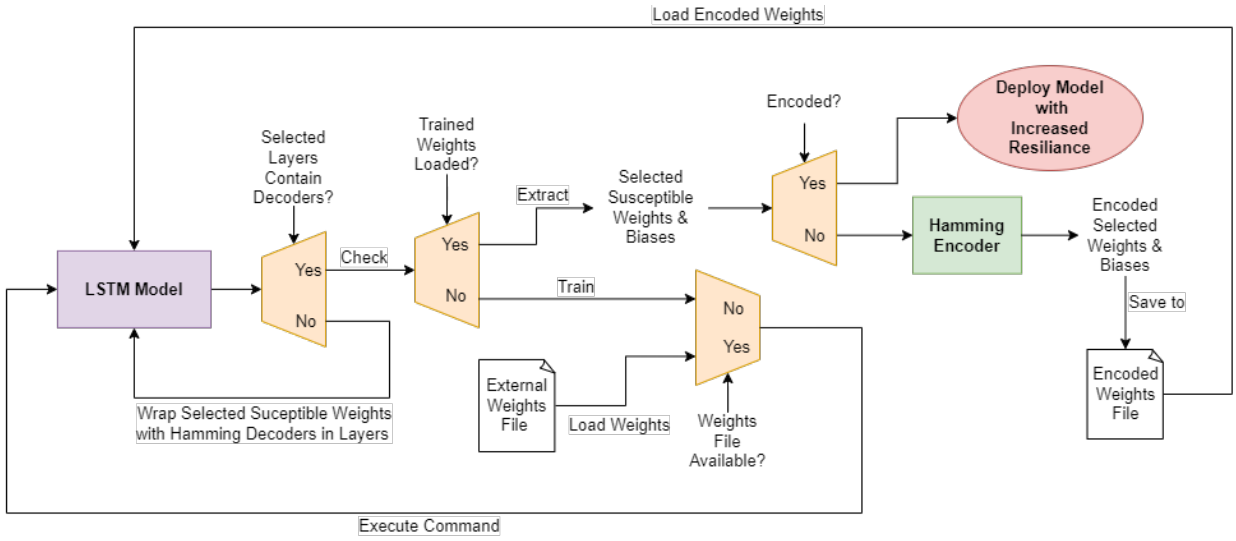


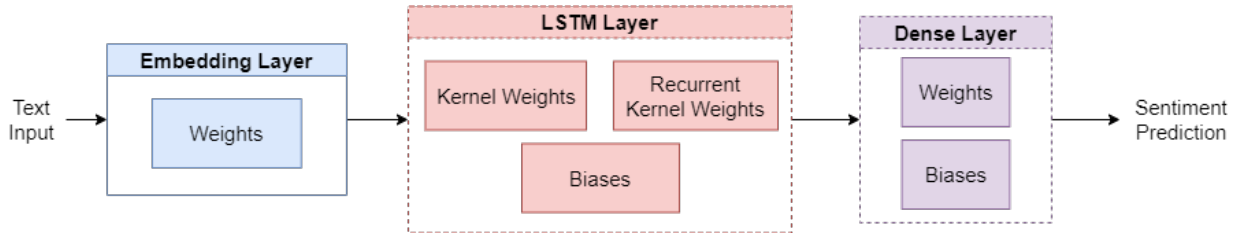
Figure 3.1. System Overview of Hamming Code for Fault Mitigation of LSTM Networks

By utilizing the error-correcting capabilities of Hamming code, we are able to harden LSTM neural networks' weights and biases, making them more resilient to SEUs of varying amounts. As further explored in this study, LSTMs show various patterns in their areas of susceptibility. By observing these common characteristics, we are able to selectively harden these training parameters, minimizing the computation needed. Our implementation of Hamming code as a mitigation method for LSTM models is depicted in Figure 3.1..

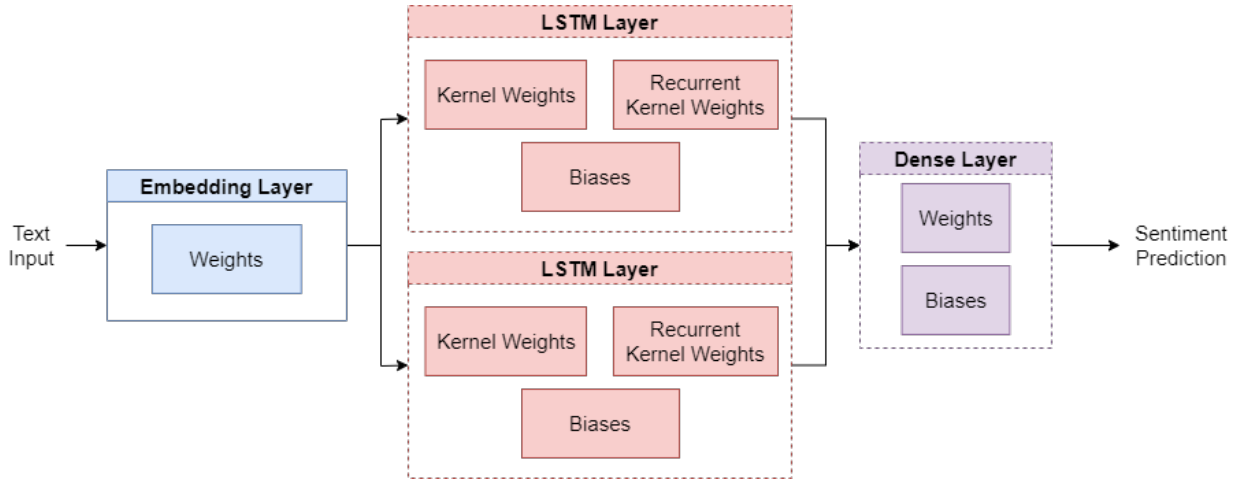
3.1. Experiment Design

3.1.1. Deep Neural Network

For the work conducted in this study, we constructed four different LSTM networks with different architectures for binary sentiment analysis with two backend frameworks, Tensorflow and Keras. Given the purpose of this neural network, all neural networks use an embedding layer as the input layer, as this will allow for the text passed to the net-

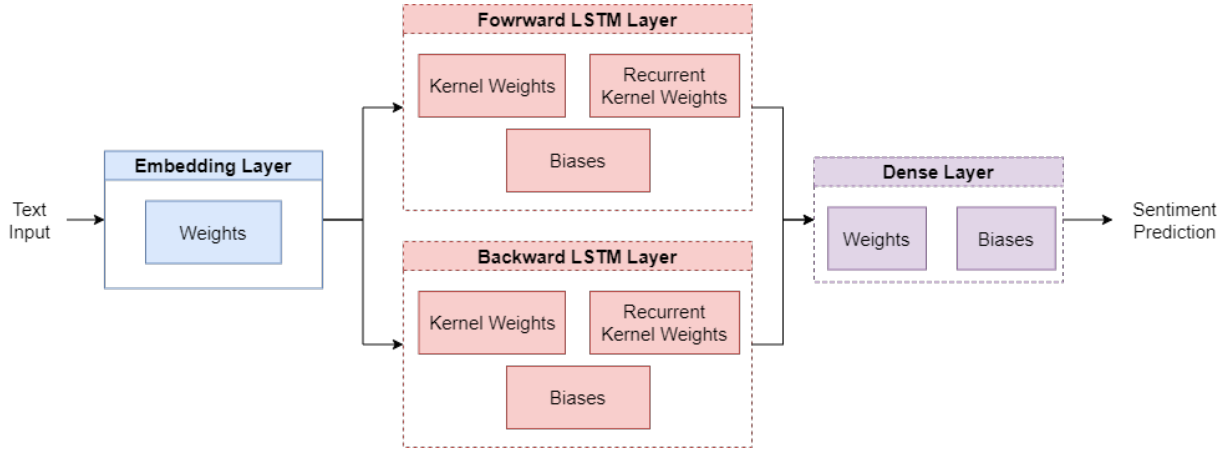


(a) Single LSTM Layer

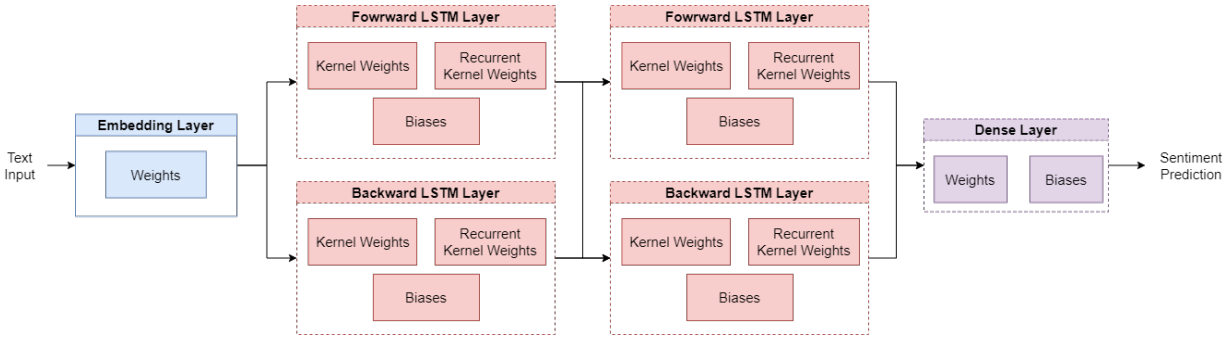


(b) Stacked LSTM Layers

Figure 3.2. LSTM Model Architectures Used for Sentiment Analysis



(c) Single Bidirectional LSTM Layer



(d) Stacked Bidirectional LSTM Layers

Figure 3.2. (cont'd) LSTM Model Architectures Used for Sentiment Analysis

work to be interpretable. This layer only consists of a single matrix of weights. For all remaining layers, save for the last layer, the architecture of each network varies. In order to cover different variations, we decided to use a single LSTM, stacked LSTMs, a bidirectional LSTM, and stacked bidirectional LSTMs for each network, respectively. As seen in Figure 3.2., the latter pair of networks will allow us to study the behavior of deeper LSTM networks with regard to faults. Additionally, the use of stacked and bidirectional LSTMs increases the overall number of weight matrices. The LSTMs are implemented via Keras's API, as they are based on the original implementation of LSTMs [12]. Lastly, all architectures contain a simple dense feed-forward layer used to output our predicted positive or negative sentiment. This layer only contains 1 matrix of weights.

3.1.2. Data Set

The data set used for training and testing with the LSTM neural networks is Stanford's Large Movie Review Data Set [17]. This data set provides 50000 highly polar movie reviews with 25000 for training and 25000 for testing. Given that this data set is intended for binary sentiment classification of movie reviews, it consists of two different classes: "positive" and "negative". This data set can be freely obtained online and is commonly used due to its large volume of data when compared to other benchmark data sets.

3.1.3. Deep Learning Framework

Implementing the LSTMs, training, testing, and preprocessing of the data set is all done with the Tensorflow [1] and Keras frameworks. This open-source framework contains all the necessary components for creating and analyzing LSTM neural networks. All parameters of each layer, such as the weights, are in single-precision floating point format

(FP32).

3.1.4. Fault Model

For our fault model, we assume that the training process is fault-free. For randomly selected weights and biases from each layer, we perform random bit-flips and observe the network's final output. This process is based on other FI methods used in prior work within this area [3][4][5][16][21].

An important aspect to note is that we consider only single-event upsets (SEUs) that would not lead to obvious failures such as a system crash. We will only consider faults that result in SDC (silent data corruption). Furthermore, this fault model simulates the SEUs or bit-flip faults which can occur in data stored within the memory of a system.

As previously mentioned, the data to be flipped consists of the weights' and biases' bits in the networks. An example of this process can be seen in Figure 3.3., where the fourth most significant bit of a given weight is flipped. An important aspect of these weights is that they are in FP32 format. Specifically, the figure depicts the original value increasing by a factor of approximately 4 billion, showing the drastic changes that can result from a single bit-flip.

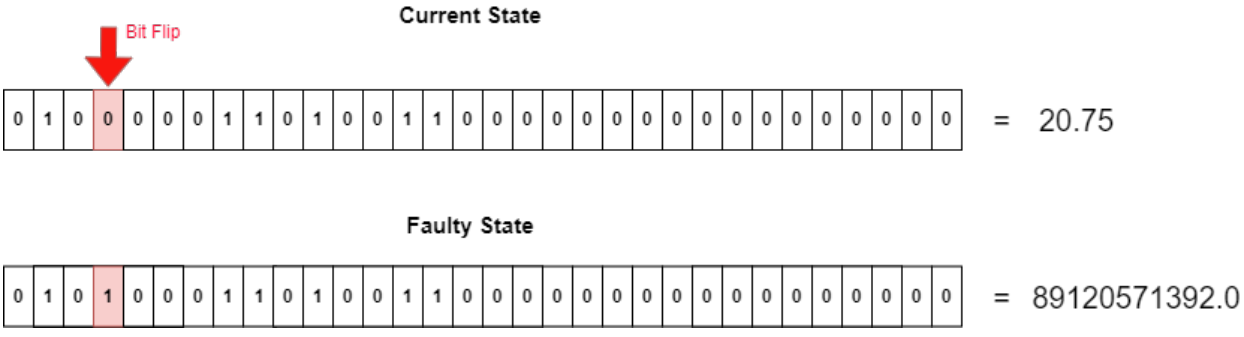


Figure 3.3. Bit-Flip on Weight in Single-Precision Floating Point Format

This model for FI chooses the layer, weights/biases, and bit for random injection.

In this way, we can effectively simulate the occurrence of bit-flip faults that could potentially affect a neural network in a given system.

Based on the model described above, we can evaluate the accuracy of the models before and after the FIs to determine the effects of specific injections or groups of injections. All FIs will occur during inference time, meaning that all weights and biases will already be set within the network. Given that we have saved the model to storage and will be loading it into memory, all injections performed on the LSTM networks will only affect the copy of the network in memory. Therefore, we can easily perform experiments on individual faults or groups of faults without needing to worry about past injections affecting the network in storage.

3.1.5. Fault Injection Framework

Interfaces

TensorFI 2 [5] is a fault injector for TensorFlow 2 applications written in Python. This injector makes use of the Keras Model APIs to inject static faults in the layer states and dynamic faults in the layer outputs; however, we will focus on injection into the layer states only. In order to utilize this fault injector, it must be imported into the Python file that contains the model to inject. Since TensorFI 2 is based on Tensorflow and Python, the model that is being evaluated must also be based on these technologies. Given that this injector only performs injection, any output files, data, and evaluation will be implemented by us in the Python file. In addition to this flexible output framework, the fault injector is configurable with YAML and command line arguments.

As shown in Figure 3.4.a, within the YAML file, we can specify whether we want

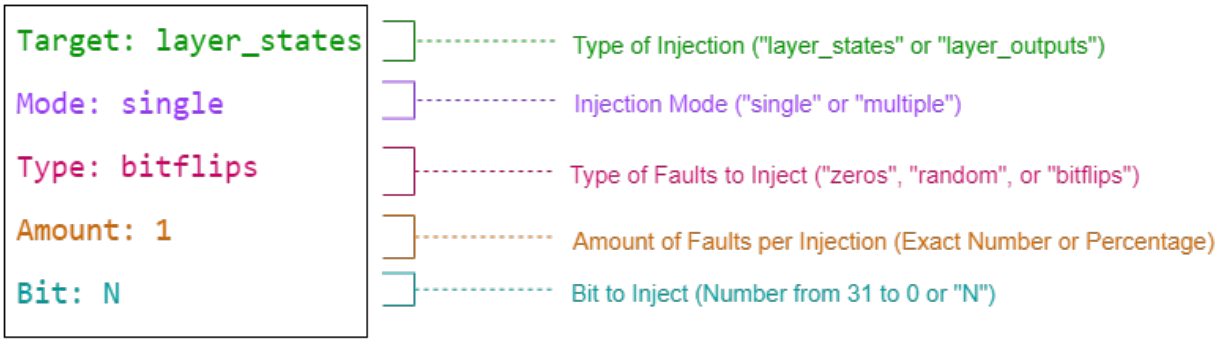
to inject into the layer states or outputs depending on the value of "Target". "Mode" will dictate whether or not we inject into a single or multiple randomly selected layers of the model. "Type" can have a value of "zeros," "random," or "bit-flips" which will specify which type of values to inject as faults. Depending on the value of the previous parameter, the "amount" parameter will specify the number of faults per injection. Lastly, "bit" can be used to inject a specific bit; however, if set to "N," a random bit will be selected for each injection.

Similarly, as seen in Figure 3.4.b, four command line arguments are passed to the Python script which contains the model to be injected. The first argument simply passes the location of the configuration YAML to the injector called in the Python file for the model. Additionally, the second argument passes the location of the directory to write any output files. In our case, we will be outputting various CSV files which will contain the error produced from each injection. The third argument specifies the number of injections to perform. Lastly, the fourth argument passes the number of samples used to evaluate the original and faulty models. These samples can be taken from any set of data, depending on what the developer chooses to pass to the injector. In our case, we will utilize the test set from Stanford's Large Movie Review Data Set as the source of all samples.

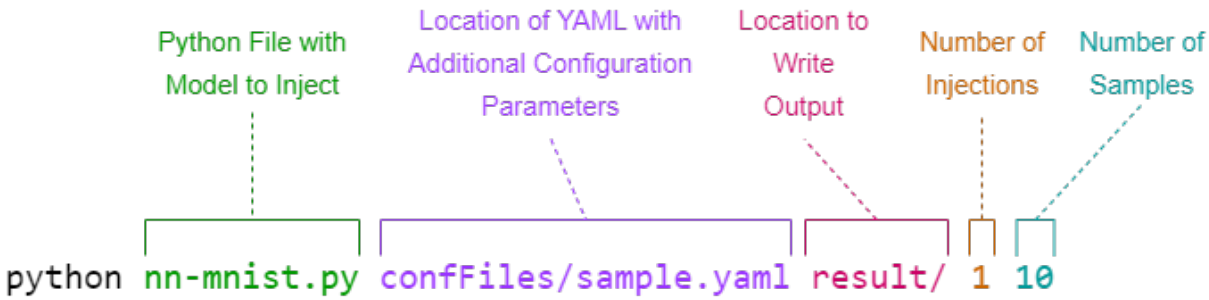
Although TensorFI 2 provides the tool for FI, users must still develop a method to utilize it and extract data from the injected model.

Functionality

TensorFI 2 performs fault injection based on the configuration parameters set in the YAML and command line arguments. In our case, the fault injector will randomly se-



(a) YAML Configuration Parameters



(b) Command for Running Fault Injector with 1 Fault Injection for 10 Different Samples

Figure 3.4. Fault Injection Configuration Options for TensorFI 2

lect a set of weights or biases in a given layer, an element within the layer (i.e. a weight), and a bit of the element to inject with a fault. For our work, all weights are assumed to be in single-precision floating point format; therefore, the randomly selected bit position will be between 31 and 0. In the case of multiple-layer injections, every set of weights and biases along with bit positions will be selected for a given injection. For both single-layer and multiple-layer injections, the number of faults per injection into the sets of weights and biases can be adjusted.

Although TensorFI 2 performs fault injection, users must still perform two preprocessing steps: 1) loading and splitting the data set and 2) loading and compiling the saved

model. Once these steps are completed users can design their own method for performing FI around TensorFI 2. Figure A.1. in the Appendix depicts the in-depth pseudocode of our method of FI analysis based on TensorFI 2. Our method consists of two rounds of inferencing in order to calculate the error present within the model after injection based on samples from the test set.

During the first round of inferencing, we pass a single sample from the test set to the original model. Given that we are only passing a single sample, we know that the accuracy must be either 0 or 100 percent. Therefore, since we only want to measure the negative impact of FIs on the model, this first round of inferencing is used for filtering out instances of classification of samples within the test set. Although a FI could potentially increase the performance of a model, this fact is beyond the scope of this work.

Between each round of inferencing, we call the injector to inject the model with faults based on the parameters previously read. Details regarding the criteria and injection process used in this work are detailed in the next section.

With the model injected with faults, the second round of inferencing is performed with the filtered test set. By comparing the results of each round of inferencing, we can determine if the FI impacts the accuracy of the model. Using the following equation

$$\text{Inaccuracy of Faulty Model} = \frac{\text{Samples Misclassified by Faulty Model}}{\text{Samples Correctly Classified by the Original Model}} \quad (3.1)$$

we calculate a score of inaccuracy (also known as error) now present within the model. This score will be saved for each pair of inferencing rounds via an exported CSV file. With a simple bash script, these CSV files can be compiled into a singular CSV file with the error produced from each individual injection.

Modifications

Despite the utility of this fault injector, we made a few modifications to it to better suit the needs of this study. In order to better locate each instance of FI and its effects, we implemented a logger into the injector. This logger generates LOG files that contain information regarding the values within the injected layer(s) before and after injection, as seen in the Appendix in Figure A.2.. This information can allow us to better see susceptibility in terms of the network's architecture, such as layers, elements, or bits more prone to SDC. With the assistance of a few simple bash scripts, information from these LOG files can easily be compiled into a single CSV file that specifies relevant data. Additionally, instead of calculating an SDC rate with a limited number of samples, we allowed the fault injector to test the faulty model with the entire test set. This allowed us to collect the accuracies of the faulty models based on the entire test set to better compare them to the original models'.

3.2. Experiments

3.2.1. Hardware & Software Specifications

The hardware used to run experiments consists of a Ryzen 5950X 16-core CPU, 64GBs of 3600MGHz RAM, and the Windows 10 operating system. For GPU-based parallelization, the local machine's NVIDA RTX 3090 GPU will be used. Additionally, Tensorflow 2.11.0 was utilized in conjunction Keras 2.11.0.

3.2.2. Experiment 1

As previously mentioned in this work, our goal of this study is to discover the susceptibility of LSTMs and develop a means of increasing their fault tolerance. Therefore, in

an effort to better understand these networks, we used 4 LSTM neural network architectures commonly used for sentiment analysis and injected each with faults, measuring their post-injection performance. Specifically, we individually inject each set of weights and biases in each layer, analyzing the effects of faults for each set. The weights and biases from each set are randomly chosen. For our study, we chose to perform 100 fault injections for each set of weights and biases in each layer. We repeated this process for 1, 10, and 20 fault injections. Once the injections were performed, we measured the accuracy of the injected model based on the 25000 sample test set. Given our weight-set approach for injection, we are able to identify both sets of weights/biases and layers of the network which are most susceptible to SEUs. Furthermore, variation in the number of faults allows us to stress-test the resilience of the network, further highlighting susceptible weight/bias sets and layers.

To continue this investigation, we also looked into the effects of injecting specific sections of the FP32 format. We chose to limit the bit-field to be injected into (i.e. the sign, exponent, and mantissa) for individual experiments while maintaining the previous parameters. This analysis will allow us to have more insight into the bits which need to be hardened within specific weight sets and layers.

3.2.3. Experiment 2

Based on our previous experiment, we developed a method to potentially improve the resilience of LSTMs. Furthermore, we made alterations to each of the LSTM models to implement this proposed method. Therefore, to measure the effectiveness of our changes, we performed 1, 10, and 20 fault injections into the most susceptible weight sets.

However, we chose to repeat injections into only all the bits of each weight and the bit-field which resulted in the most SDC. Given that these modifications may result in additional computation, we measured the execution time of each model on the testing set. By hardening the models based on the previous experiments, and comparing their performance to the original models, we hope to introduce a potentially low-cost, solution for hardening the weights of these neural network models.

Chapter 4. Results & Analysis

4.1. Training Results

The LSTM models were trained on Stanford's Large Movie Review Data Set with the Adam optimizer and a learning rate of 0.001 with a batch size of 128. Tensorflow and Keras were the only packages used to perform data preprocessing and loading along with building and training the LSTM models. The accuracies of the models after training each for 5 epochs can be seen in Table 4.1..

Table 4.1. Accuracies of the LSTM Networks

Model	Accuracy (%)
LSTM	87.10
Stacked LSTM	86.73
Bidirectional LSTM	86.89
Stacked Bidirectional LSTM	87.81

4.2. Statistics of Network Parameters

Given that we will be looking at the weights and layers of multiple neural networks, it is important to know the number of weights and biases within each layer. Table 4.2. details the different sets of weights and biases and each model's respective layers.

In addition to the number of nodes, knowing the *fault percentage*–which indicates the ratio between the number of faulty weights and the total number of weights–can give a better perspective on the number of faults present in a given layer [18]. Table 4.3. details these fault percentages for different numbers of weights in the LSTM models. Given that some latter weight sets possess smaller numbers of weights, one may expect these to be the most impacted by FI.

Given that the faults will be injected into trained weights, observing the distribu-

Table 4.2. Distibution of Weights and Biases in Each LSTM Model

Layer	Weights/Biases	Number of Weights/Biases
Embedding	Kernel Weights	75000
LSTM	Kernel Weights	600
	Recurrent Kernel Weights	400
	Biases	40
Dense	Kernel Weights	10
	Biases	1
Total		76,051

(a) Single LSTM Layer

Layer	Weights/Biases	Number of Weights/Biases
Embedding	Kernel Weights	75000
LSTM	Kernel Weights	600
	Recurrent Kernel Weights	400
	Biases	40
LSTM	Kernel Weights	400
	Recurrent Kernel Weights	400
	Biases	40
Dense	Kernel Weights	10
	Biases	1
Total		76,891

(b) Stacked LSTM Layer

tion of these weights before injection can give some insight into why FI might introduce destructive behavior. As shown in Figure 4.1., the weights of the LSTM models are distributed in the range of [-1.0, 1.0] with the majority of weights being between [-0.50, 0.50].

If we consider that the range of FP32 numbers is approximately 10^{-38} to 10^{+38}, we can see that, in the context of the LSTM model, this number format greatly exceeds the required range of the model's weights. As we will see in further analyses, the availability of such a wide range of values can lead to drastic weight changes in the network when

Table 4.2. (cont'd) Number of Weights and Biases in Each Layer within Each LSTM Model

Layer	Weights/Biases	Number of Weights/Biases
Embedding	Kernel Weights	75000
Forward LSTM	Kernel Weights	600
	Recurrent Kernel Weights	400
	Biases	40
Backward LSTM	Kernel Weights	600
	Recurrent Kernel Weights	400
	Biases	40
Dense	Kernel Weights	20
	Biases	1
Total		77,101

(c) Bidirectional LSTM Layer

Layer	Weights/Biases	Number of Weights/Biases
Embedding	Kernel Weights	75000
Forward LSTM	Kernel Weights	600
	Recurrent Kernel Weights	400
	Biases	40
Backward LSTM	Kernel Weights	600
	Recurrent Kernel Weights	400
	Biases	40
Forward LSTM	Kernel Weights	800
	Recurrent Kernel Weights	400
	Biases	40
Backward LSTM	Kernel Weights	800
	Recurrent Kernel Weights	400
	Biases	40
Dense	Kernel Weights	20
	Biases	1
Total		79,581

(d) Stacked Bidirectional LSTM Layer

Table 4.3. Fault Percentages for Different Number of Weights Based on LSTM Models

Number of Weights	Fault Percentage (%)		
	1 Fault	10 Faults	20 Faults
75000	0.001	0.013	0.026
800	0.125	1.250	2.5
600	0.167	1.667	3.333
400	2.5	25	50
40	10	100	N/A
20	5	50	100
10	10	100	N/A
1	100	N/A	N/A

faults are introduced.

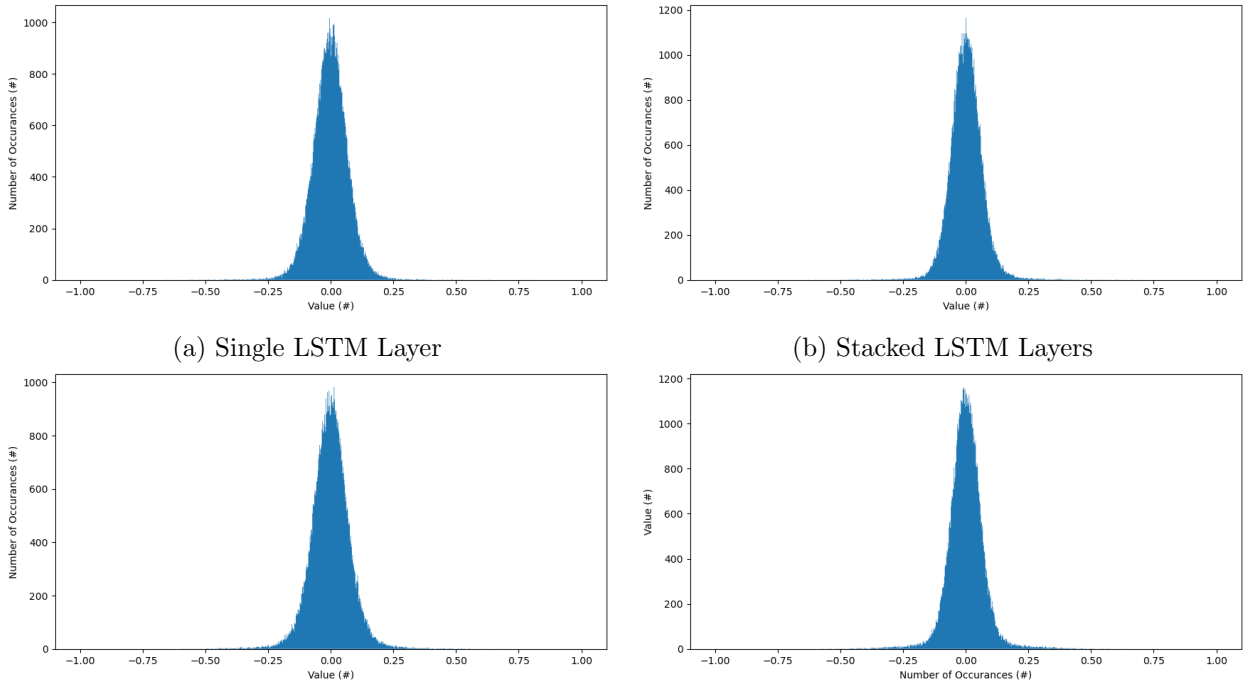
4.3. Criteria for TensorFI2's Parameters

As previously mentioned, TensorFI 2 heavily relies on a variety of parameters, making it quite flexible; therefore, we detail the parameters used for all experiments in this work in Table 4.4..

4.4. Experiment 1: Investigation of LSTM Models' Resilience

4.4.1. Single Event Upset (SEU) Injections Results

As previously mentioned, we measured the resilience of each of the 4 models by injecting SEUs into each weight/bias set of each layer within the networks. We performed



(a) Single LSTM Layer

(b) Stacked LSTM Layers

(c) Single Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

Figure 4.1. Distribution of Trained Weights & Biases of the LSTM Networks

injection over 100 iterations for each weight/bias set with 1, 10, and 20 SEUs per iteration. Given the vast number of tests performed for each model, we calculated the average accuracy of each 100 iterations for each weight/bias set and compiled the results into Tables 4.5. - 4.8.. Note that some tests could not be performed due to the number of weights/biases in certain layers of each network; therefore, values of "N/A" are written in the tables to signify these instances. Additionally, Figures B.1. - B.12. in the Appendix show the individual results of these injections, i.e. where we derived our average accuracies.

Table 4.5. depicts the neural network with only a single LSTM layer. This model is the simplest and least deep of the four. A noticeable observation of this model's performance under faults is its resilience to them in the first layer, i.e. the embedding layer. Given that this layer possesses the most amount of weight at 75000, there is no surprise

Table 4.4. Configuration Parameters Used with TensorFI2 for First FI Experiment

Location	Configuration Parameters	Value	Description
YAML File	Target	layer_states	Set whether to inject into the layer states or layer outputs
YAML File	Mode	single	Determine whether to inject faults into a single or multiple layer(s)
YAML File	Type	bitflips	Set the type of fault to inject (i.e. zeroes, random, or bitflips)
YAML File	Amount	1, 10, 20	Set number of faults to inject per injection
YAML File	Bit	N	Specify bit position to inject ("N" signifies a random position)
Command Line Argument	"Number of Injections"	100	Determines number of injections to perform
Command Line Argument	"Number of Samples"	25000	Determines number of samples to test with per injection

that this layer did not see any noticeable changes in its performance when injected with 1, 10 or even 20 SEUs in various bit-fields. This behavior of this specific layer will continue throughout all the models. Therefore, we will not discuss it in subsequent models. Moving onto the LSTM layer's weights and biases, we noticed that all sets were susceptible to SEUs, especially when they occurred in the exponent bits. As expected, larger amounts of SEUs gave rise to worse accuracies most of the time. Under 20 SEUs in the exponent bit-field, the average accuracy of the biases was 56.45. Similar trends in accuracy drops could also be seen when the range of injectable bits was 0 to 31. It is worth noting that negligible change was observed in the accuracies when the mantissa bit-field was injected. However, for the biases, the sign bit-field appeared to be a susceptible location with higher

Table 4.5. Matricies of the Average Accuracies of Single LSTM Layer Model After 1, 10, and 20 SEUs Over 100 Iterations

Injected Bit Field & Number of Injections Performed per Iteration			Injected Layer & Weight/Bias Set					
			Emb.	LSTM			Dense	
			KW	KW	RKW	B	KW	B
	All Bits	1	86.98	86.87	86.98	86.20	87.05	85.61
		10	87.07	86.25	81.70	79.59	82.12	75.25
		20	86.63	84.07	79.77	73.05	77.79	61.91
	Exponent Bits	1	87.10	86.26	85.21	82.59	85.19	83.00
		10	87.02	83.04	73.48	65.57	77.97	N/A
		20	86.80	78.36	68.67	56.45	80.60	N/A
	Mantissa Bits	1	87.10	87.10	87.10	87.10	87.10	87.10
		10	87.10	87.10	87.08	87.02	87.09	87.10
		20	87.10	87.10	87.09	86.95	87.09	87.09
	Sign Bit	1	87.10	87.08	86.91	85.90	87.01	87.17
		10	87.10	87.02	82.62	77.15	12.83	N/A
		20	87.09	86.90	81.71	71.95	N/A	N/A

amounts of SEUs (i.e. 10 and 20 in this case). Based on these observations, the LSTM layer's biases appear to be the most susceptible of these three weight sets. If we consider that the LSTM layer possesses only 40 biases, 10 and 20 SEUs in each bias result in fault percentages of 25% and 50%, explaining why faults drastically affect this set. In the final layer of the network, i.e. the dense layer, we observed some instances of SDC similar to the LSTM layer. This layer of the network contains the least amount of weights and biases at 10 and 1, respectively. Therefore, fault percentages of this layer were higher than most

of the other weight/bias sets. In fact, any single SEU in the bias of the dense layer results in a fault percentage of 100%. With this in mind, we can see that the exponent bit field appears to be the only bit field that drastically affected this weight. This same fact can be said for the kernel weights. We did observe the lowest accuracy of all our injection tests on this model at 12.83%. This accuracy occurred for 10 injections in the sign bits. However, just as with the dense layer's biases, the fault percentage of this particular test is 100% due to the small number of weights. This small number of weights and biases in this layer makes it intrinsically susceptible to SEUs, as each fault incurs a higher effect than in other layers. Considering all the various layers of this network, the LSTM and dense layers are the most susceptible.

The second model used for injection is similar to the previous; however, instead of a single LSTM layer, this network contains two consecutive LSTM layers. Among all the layers of the network, the first LSTM layer, the dense layer and the biases of the second LSTM layer appeared to be the most susceptible to SEUs. Similar to the previous model, both the LSTM layer of the former model and the LSTM layer of this model possessed their lowest accuracies when the SEUs were in the exponent bit-field. Additionally, the biases' behavior to larger amounts of SEUs in the sign bit-field remains consistent. However, when compared to the first model, this model's first LSTM layer appears to contain much worse accuracies. Interestingly, the second LSTM layer possessed very little SDC when the kernel weights and recurrent kernel weights were injected. Only the biases of this layer saw any notable change when the exponent bit field was injected with 10 and 20 SEUs. This behavior differs from the first model in that its accuracy, when all the bits were injected in the biases, was much higher. The dense layer seemed to maintain the same behavior that

Table 4.6. Matricies of the Average Accuracies of Stacked LSTM Layers Model After 1, 10, and 20 SEUs Over 100 Iterations

Injected Bit Field & Number of Injections Performed per Iteration			Injected Layer & Weight/Bias Set								
			Emb.	LSTM 1			LSTM 2			Dense	
			KW	KW	RKW	B	KW	RKW	B	KW	B
All Bits		1	86.71	86.35	86.11	86.49	86.72	86.72	86.72	86.72	84.87
		10	86.69	84.25	82.59	79.18	86.71	86.71	84.44	83.78	78.16
		20	86.66	84.52	79.06	69.45	86.68	86.57	82.90	80.23	62.67
Exponent Bits		1	86.72	85.93	85.37	85.19	86.71	86.69	85.95	85.54	80.80
		10	86.48	81.76	73.85	66.21	86.19	86.60	76.89	81.98	N/A
		20	86.11	76.98	64.67	55.75	86.04	84.83	70.19	80.68	N/A
Mantissa Bits		1	86.72	86.72	86.72	86.71	86.72	86.72	86.72	86.72	86.72
		10	86.72	86.72	86.72	86.67	86.72	86.72	86.72	86.72	86.71
		20	86.73	86.72	86.71	86.26	86.73	86.73	86.72	86.72	86.70
Sign Bit		1	86.72	86.72	86.47	85.68	86.72	86.72	86.72	86.71	86.72
		10	86.72	86.66	84.19	78.01	86.72	86.70	86.71	13.28	N/A
		20	86.72	86.62	82.17	73.25	86.71	86.68	86.74	N/A	N/A

it possessed in the previous model due to its small number of weights. For the most part, this model is extremely similar to the previous model in its behavior to SEUs.

The third model consists of a single bidirectional LSTM layer. Bidirectional LSTMs allow information to flow both forward and backward, explaining why the LSTMs are referred to as forward and backward LSTMs layers. Looking at the model's injection results, they are comparable to the previous model's results in that the first LSTM layer, i.e. the forward LSTM, and the dense layer appear to be the most affected. Additionally,

Table 4.7. Matricies of the Average Accuracies of Bidirectional LSTM Layer Model After 1, 10, and 20 SEUs Over 100 Iterations

			Injected Layer & Weight/Bias Set								
			Emb.	Forward LSTM 1			Backward LSTM 1			Dense	
			KW	KW	RKW	B	KW	RKW	B	KW	B
Injected Bit Field & Number of Injections Performed per Iteration	All Bits	1	86.88	86.70	86.45	84.76	86.88	86.89	86.89	86.15	86.48
		10	86.71	84.64	81.14	76.28	86.82	86.83	84.27	79.24	77.15
		20	86.83	82.97	75.66	69.63	86.78	86.79	82.82	77.64	66.66
	Exponent Bits	1	86.88	85.38	84.95	82.84	86.87	86.87	86.14	83.88	82.03
		10	86.77	79.78	71.12	65.39	86.72	86.75	80.16	70.15	N/A
		20	86.53	73.68	66.67	58.64	86.56	86.53	71.30	64.24	N/A
	Mantissa Bits	1	86.88	86.89	86.88	86.83	86.88	86.88	86.89	86.89	86.89
		10	86.89	86.89	86.87	86.27	86.89	86.89	86.90	86.89	86.91
		20	86.88	86.89	86.82	85.43	86.89	86.89	86.90	86.89	86.91
	Sign Bit	1	86.88	86.89	86.50	85.23	86.89	86.89	86.84	86.77	86.76
		10	86.89	86.71	79.53	61.08	86.89	86.81	86.11	47.33	N/A
		20	86.89	86.25	74.77	51.86	86.89	86.69	85.16	13.24	N/A

most instances of inaccuracy result from injections into the exponent bit-field. Comparing the severity of the third and second model's results, we can see that both have around the same range of inaccuracies in each layer. Again, biases continue to be the most sensitive to SEUs, especially those in the dense and forward LSTM layers.

The fourth and final model is known as a stacked bidirectional LSTM, combining all the ideas of the previous three models. This network possesses 15 different sets of weights and biases, making it the most complex in our case of testing. Despite the in-

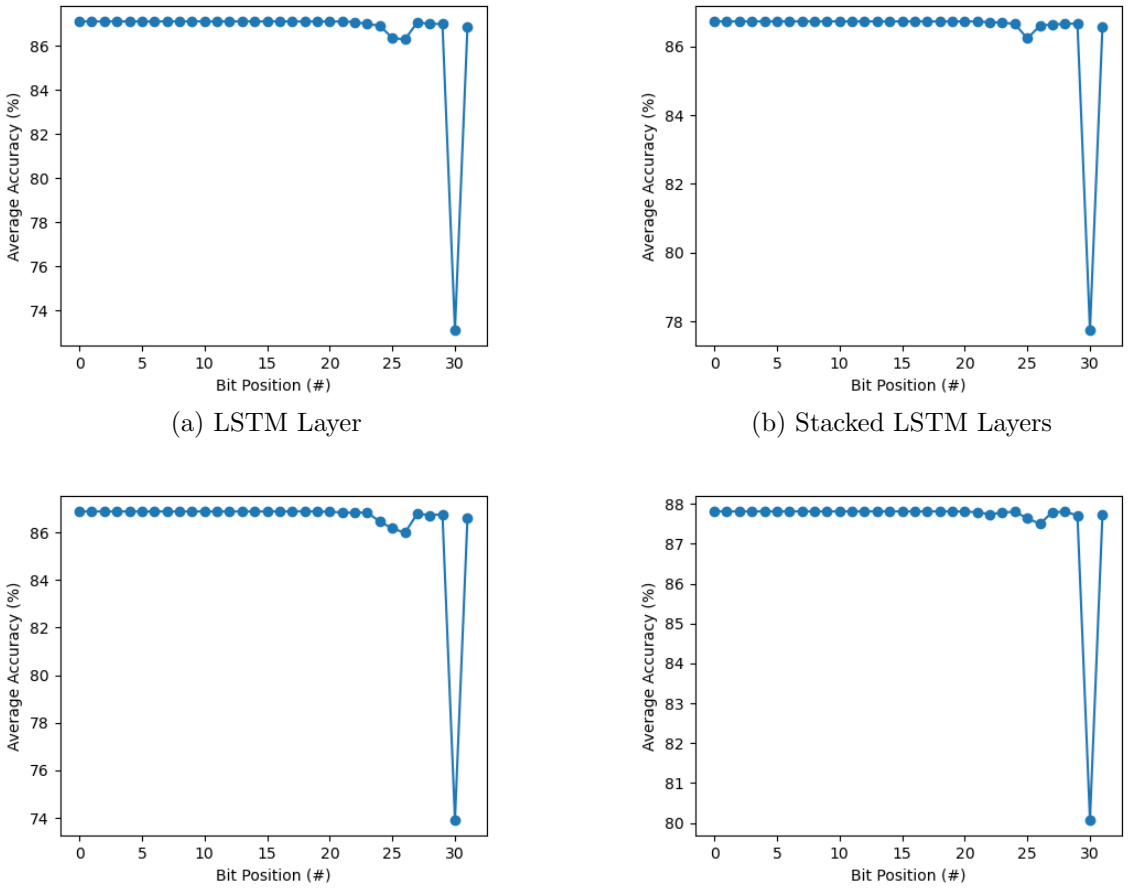
Table 4.8. Matricies of the Average Accuracies of Stacked Bidirectional LSTM Layer Model After 1, 10, and 20 SEUs Over 100 Iterations

Injected Bit Field & Number of Injections Performed per Iteration			Injected Layer & Weight/Bias Set														
			Emb.	Forward LSTM 1			Backward LSTM 1			Forward LSTM 2			Backward LSTM 2			Dense	
			KW	KW	RKW	B	KW	RKW	B	KW	RKW	B	KW	RKW	B	KW	B
	All Bits	1	87.81	87.81	87.67	86.57	87.81	87.71	87.71	87.81	87.81	87.82	87.81	87.81	87.43	87.21	86.30
		10	87.78	86.68	83.58	80.33	87.74	87.36	86.99	87.77	87.76	85.52	87.81	87.82	84.79	83.66	74.58
		20	87.74	85.66	78.38	74.30	87.65	87.22	81.13	87.75	87.64	83.57	87.81	87.81	81.38	80.35	65.12
	Exponent Bits	1	87.79	87.36	85.86	84.81	87.77	87.76	86.88	87.80	87.81	85.90	87.81	87.81	86.68	85.74	85.17
		10	87.51	82.74	75.81	63.38	87.54	86.32	75.16	87.47	87.42	80.45	87.81	87.81	81.76	75.49	N/A
		20	87.42	79.32	70.54	59.22	87.14	86.50	69.66	87.21	86.28	74.23	87.42	87.42	76.08	71.31	N/A
	Mantissa Bits	1	87.81	87.81	87.81	87.73	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.81
		10	87.81	87.81	87.81	87.57	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.81	87.82	87.82
		20	87.82	87.82	87.78	87.26	87.82	87.82	87.81	87.82	87.82	87.82	87.82	87.82	87.82	87.82	87.83
	Sign Bit	1	87.81	87.82	87.44	86.99	87.81	87.81	87.80	87.81	87.82	87.82	87.81	87.81	87.81	87.80	87.80
		10	87.81	87.78	84.98	79.20	87.81	87.79	87.48	87.81	87.80	87.80	87.82	87.82	87.80	46.80	N/A
		20	87.81	87.77	83.02	74.77	87.81	87.78	87.14	87.79	87.79	87.72	87.82	87.82	87.76	12.20	N/A

creased complexity, the network possesses a similar trend in behavior when SEUs are introduced. Specifically, the second forward LSTM layer and the second backward LSTM layer experience minimal SDC, maintaining average accuracies above 80%, except for when 20 SEUs are introduced within the exponent bit field range. We can consider these two layers to be the least susceptible. As we know, the dense layer is inherently susceptible due to its being the output layer with a minimal amount of weight. Turning our attention to the first forward and backward LSTM layers, we observed relatively high resilience in the latter layer, except in the case of biases being injected with multiple SEUs in the ex-

ponent bit-field. Like the other models, the first layer, excluding the embedding layer, is the most susceptible layer, especially when injected in the exponent bit field. Accuracies within the model's "first" layer appear to be similar when compared to the other three models.

4.4.2. Analysis of SEU Injections



(a) LSTM Layer

(b) Stacked LSTM Layers

(c) Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

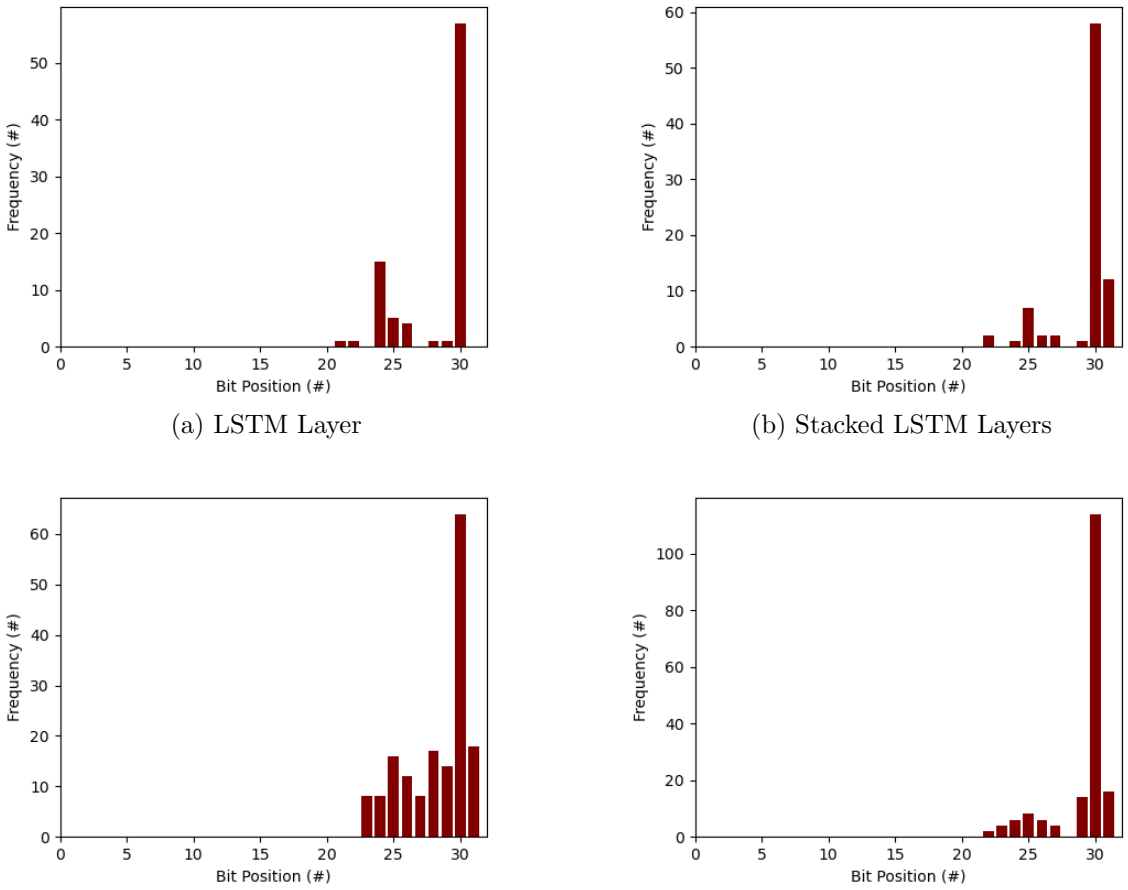
Figure 4.2. Average Accuracy of Each Bit Position Based on All Single SEU Injections Performed on Each Model

Based on the observations made from the SEU injection experiments, we have noticed that the large number of weights within the embedding layer makes it inherently resilient to our SEU tests. However, the subsequent layer of the embedding layer tends to be

most susceptible to faults within all models. Furthermore, biases generally undergo SDC when SEUs are introduced in the exponent bit field, especially when multiple are introduced. Lastly, since the dense layer contains a small number of weights and acts as the output layer, this layer is extremely sensitive to SEUs and is almost always susceptible to SEUs. More specifically, the dense layer seems to only be majorly affected when multiple faults are introduced in the exponent bit field or when multiple weights or biases experience SEUs in their sign bits. Given that we are performing binary classification, the sensitivity of the sign bits is to be expected, as the sign of the output determines the prediction outcome.

Although we determined that the exponent bit field is a common area of susceptibility, we wanted to look further into which bits specifically lead to the most SDC. Therefore, as seen through Figure 4.6., we looked at the average accuracy of each bit position for each model based on all previous single SEU injections. As expected, we can clearly see that bit positions 23 through 30, i.e. the exponent bit field, yield the most SDC when exposed to SEUs. However, bit 30 notably led to the most drastic decreases in accuracy, being at around 70% to 80% within all models. This behavior of bit 30 can be explained by the extreme change value with respect to the current distribution of weights and biases. As seen in Figure 4.4., flipping bit 30 results in small weights and biases becoming exponentially bigger. Given such a large difference in value, this weight/bias will overtake all others, resulting in SDC.

In addition to exponentially increasing the weights and biases, flipping bit 30 can also potentially result in a value of "Nan." To be more specific, this value results when all bits of the exponent are set to 1. A value of Not a number (Nan) can be detrimental to



(a) LSTM Layer

(b) Stacked LSTM Layers

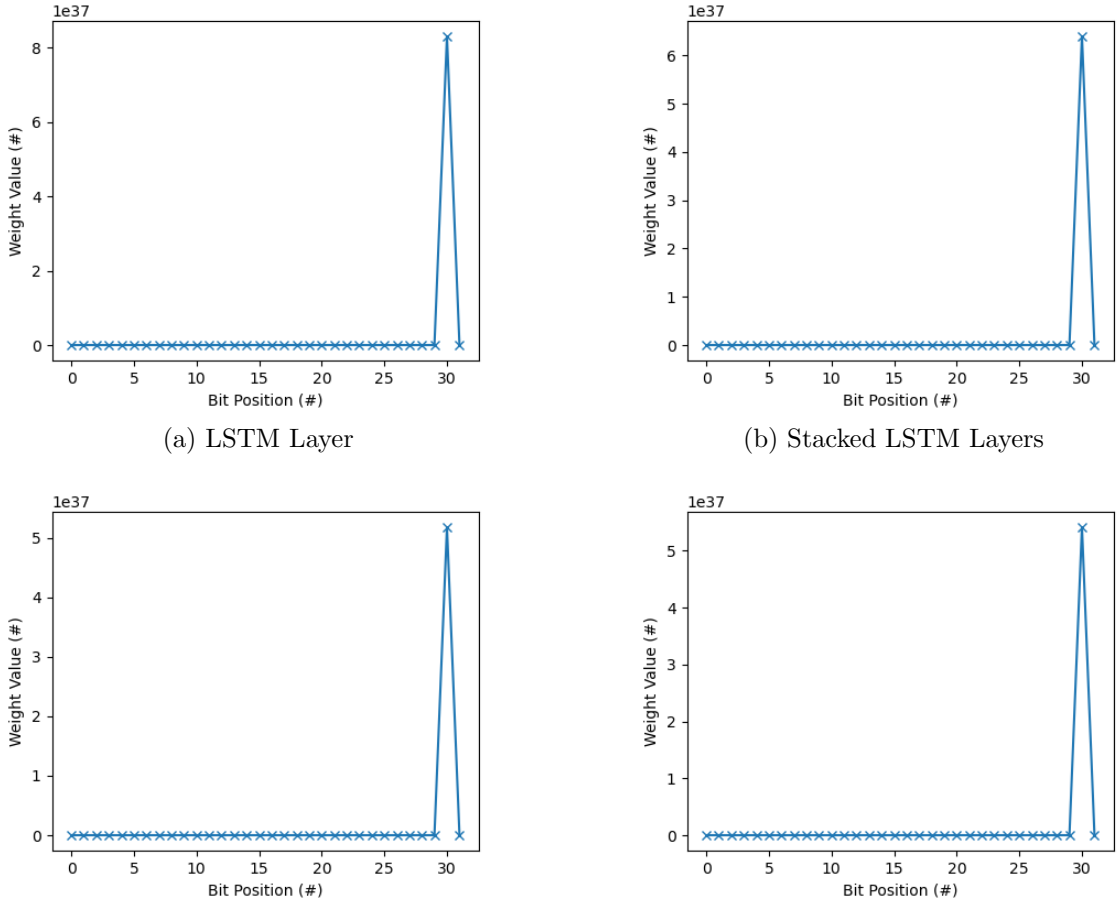
(c) Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

Figure 4.3. Frequency of Each Bit Position Resulting in SDC Based on All Single SEU Injections Performed on Each Model

the network because it has no way of being interpreted by the model, leading to its propagation to subsequent layers. As seen by Figure 4.5., this value can occur relatively frequently even with only a single SEU injection into the networks.

Based on the observations made regarding the exponent bit-field, the usage of FP32 for the weights and biases leads to susceptibility. Figure 4.6. shows the distribution of exponent values of all weights and biases in the network. This figure confirms that a majority of the 64 different exponent combinations are not being utilized. Therefore, allowing such a large range of numbers for such a susceptible bit field only increases the chances of



(a) LSTM Layer

(b) Stacked LSTM Layers

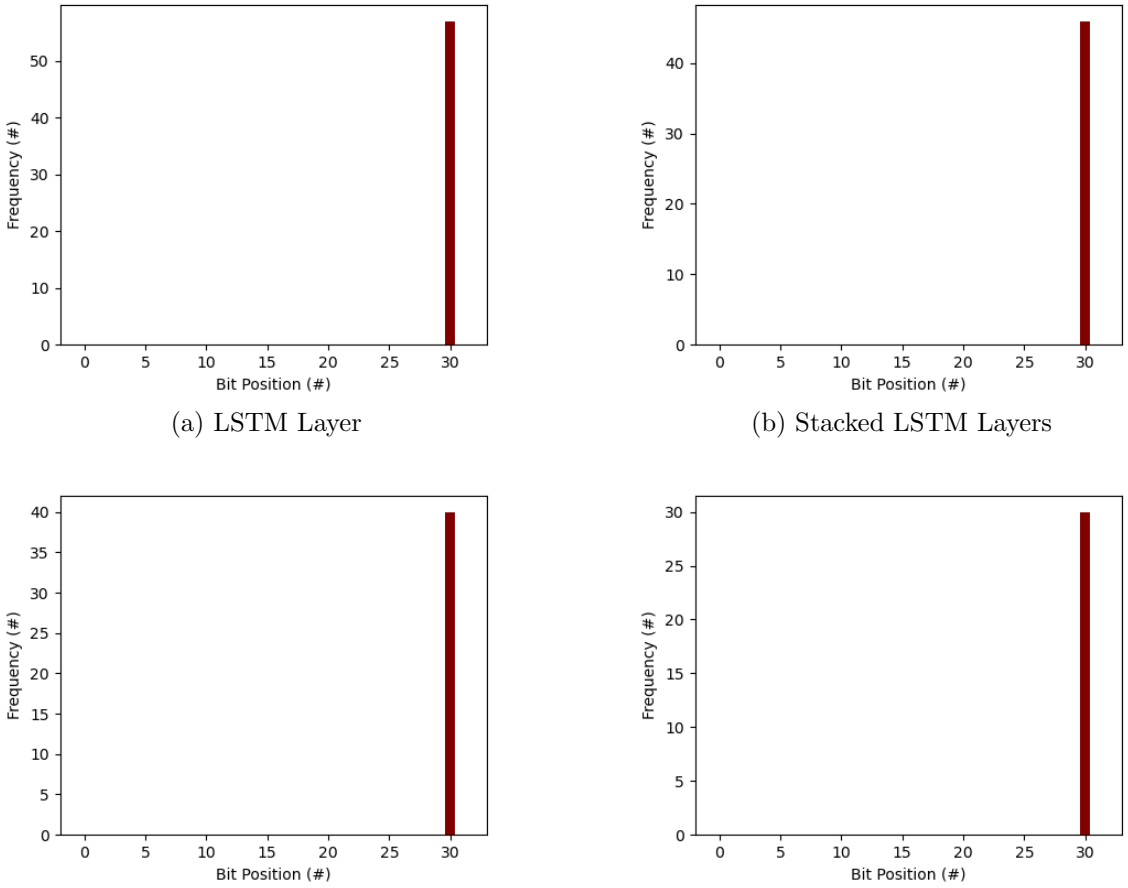
(c) Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

Figure 4.4. Average Value of Weights and Biases when Single Bit Position is Injected with an SEU (Based on Absolute Values of Weights/Biases and All Single SEU Injections Performed on Each Model)

SDC occurring within the network.

Based on the previous fault injection experiments performed, we designed a mitigation method that involves encoding susceptible bits of susceptible sets of weights and biases in each network. Our encoding and decoding is based on hamming codes which allow us to encode the most susceptible part of all FP32 weights and biases, regardless of layer: the exponent bits.



(a) LSTM Layer

(b) Stacked LSTM Layers

(c) Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

Figure 4.5. Frequency of Each Bit Position Resulting in a "Nan" Value Based on All Single SEU Injections Performed on Each Model

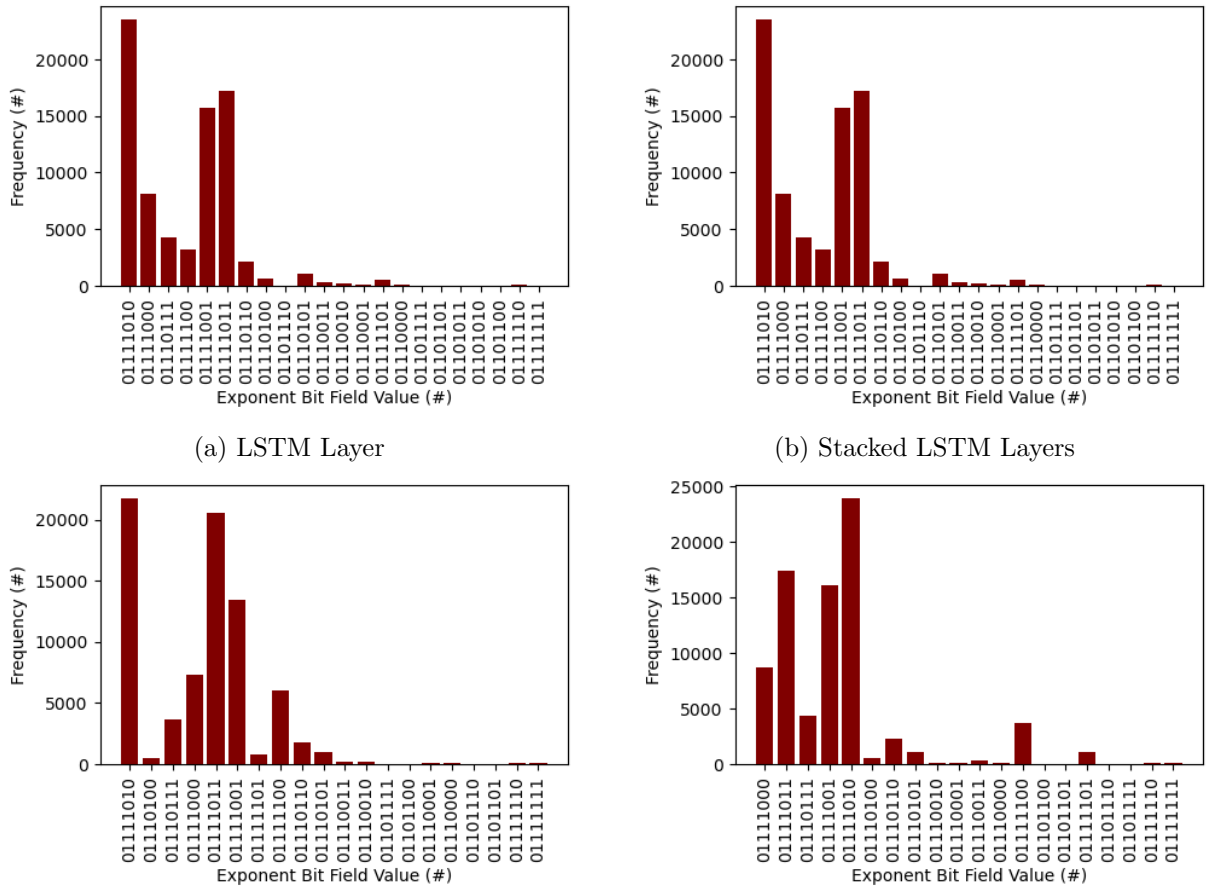
4.5. Experiment 2: Fault Mitigation for LSTM Models

4.5.1. Hamming Code & Fault Mitigation

Parity Bit Calculations

Hamming codes are a nontrivial family of error-correcting codes that can detect and correct errors that may occur during data transmission or storage [11]. Created by Richard Hamming in the 1950s, these codes are linear block codes that are based on parity-check and generator matrices.

To implement these codes, (p) parity bits needed for d data bits must first be de-



(a) LSTM Layer

(b) Stacked LSTM Layers

(c) Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

Figure 4.6. Distribution of Exponent Bit Field Value for All Weights & Biases within Each Model

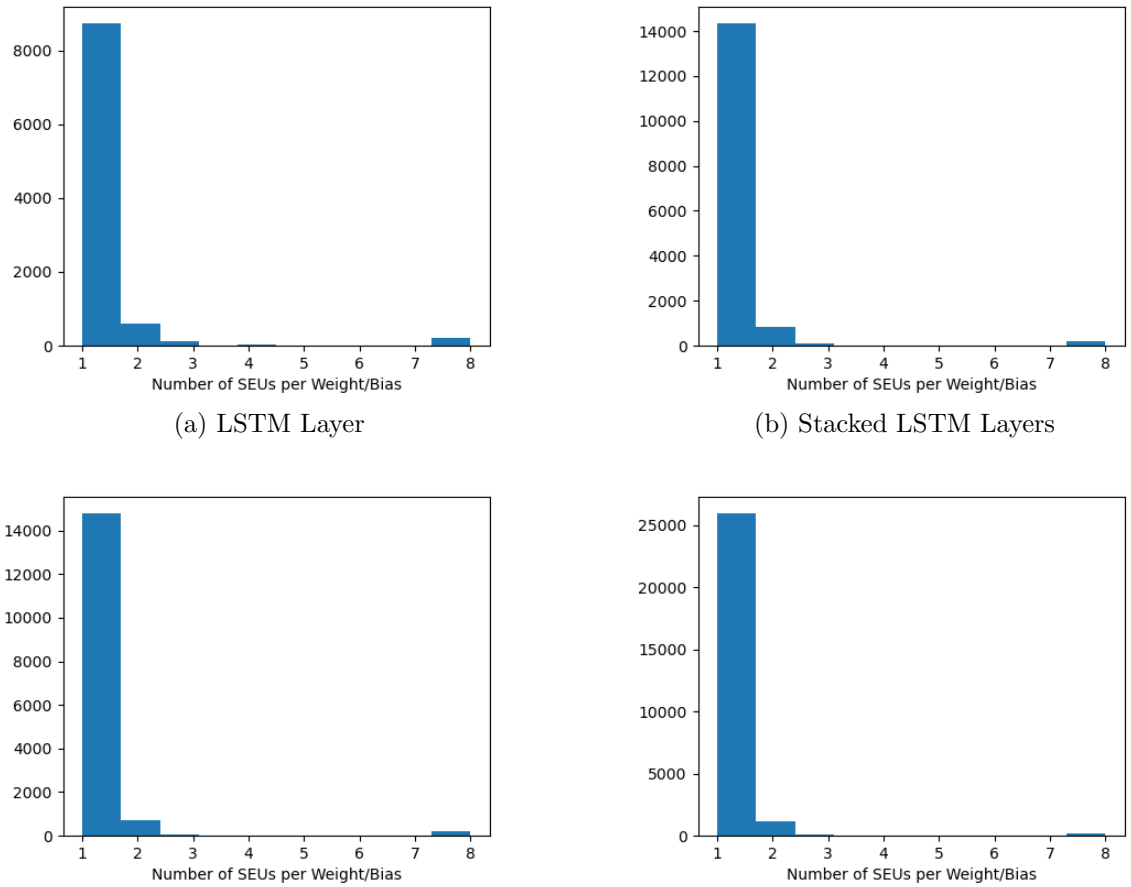
termined to define a code word length. For this task, we use equation 4.1 where p is the number of parity bits, and b is the number of bits in the data.

$$2^p \geq d + p + 1 \tag{4.1}$$

With $p = 3$, we get the most basic form of (7, 4) binary Hamming code. This encoder accepts 4-bit long information and adds 3 parity bits to it, producing 7-bits wide Hamming encoded blocks.

In our case, we wanted to encode the 8 data bits of the exponent bit-field; there-

fore, according to the equation 4.1, we would need 4 parity bits. This encoding would only allow for the correction of a single error in a weight or bias. Another means of encoding would involve splitting the exponent bit field into an upper and lower half, allowing us to encode each 4 data bits with 3 parity bits. This encoding would result in code words of length 7 for each half and a total of 6 parity bits. Additionally, it would allow us to detect a total of two errors in the exponent bit field for a single weight/bias.



(a) LSTM Layer

(b) Stacked LSTM Layers

(c) Bidirectional LSTM Layer

(d) Stacked Bidirectional LSTM Layers

Figure 4.7. Frequency of Different Numbers of SEUs Occurring within the Same Weights (Based on Injections of 10 & 20 SEUs in Exponent Bit Field)

Both methods would suffice; however, we looked at the frequency of multiple SEUs occurring in the same weight based on our previously conducted fault injection experi-

ments. Specifically, we looked at the injections with 10 and 20 SEUs per iteration. Figure 4.7. shows the distribution of different amounts of SEUs occurring in the same weight. Across all the networks, the frequency of 3 or more SEUs occurring in the same weights was significantly smaller than 1 or 2 SEUs in a single weight. Therefore, we decided to divide the weight into two 4-bit halves, as seen in Figure 4.8..

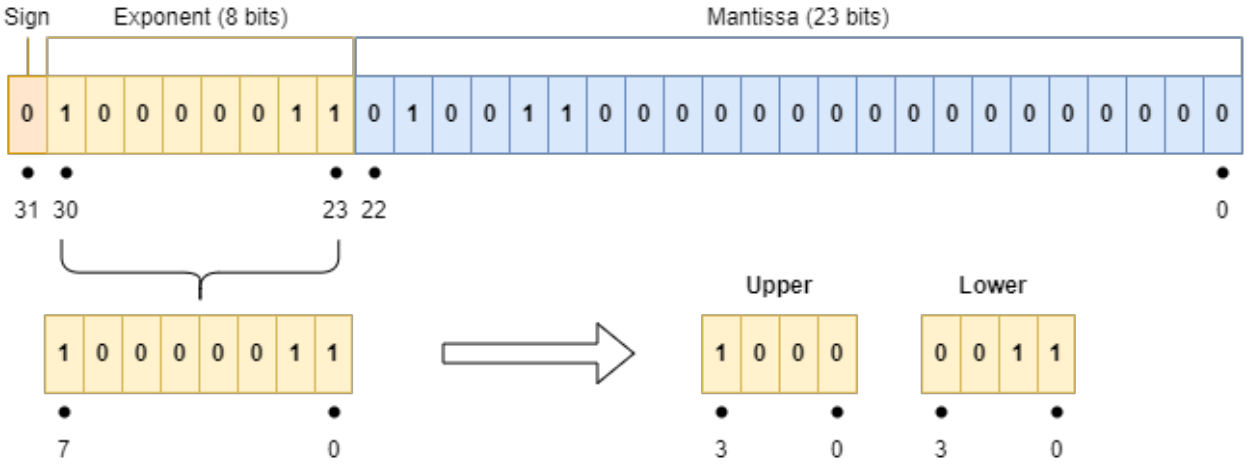


Figure 4.8. Example of Splitting the Exponent Bit field into Upper and Lower Halves

To calculate the parity bits, the XOR operation is performed on the individual data bits. Equations 4.2 show the parity bit equations that we derived. In these equations, P_1, P_2, and P_3 are the parity bits, and D_1, D_2, D_3, and D_4 are the data bits.

$$\begin{aligned} P_1 &= D_1 \oplus D_2 \oplus D_4 \\ P_2 &= D_1 \oplus D_3 \oplus D_4 \\ P_2 &= D_2 \oplus D_3 \oplus D_4 \end{aligned} \tag{4.2}$$

Hamming Encoding

With the parity bits calculated, data can be encoded into a code word by interleaving the data bits with the parity bits. Assuming that we are using (7, 4) hamming code, a code word may be represented as $[P_1, P_2, D_1, P_3, D_2, D_3, D_4]$. Different variations of this interleaving can be performed and each will result in a different form of encoding with respect to the code words. Ideally, parity bits should be placed at powers of two in the code word to allow for efficient detection and correction of errors. We can begin constructing the generator matrix, which will allow us to encode any 4-bit data according to our hamming code scheme.

In order to derive this matrix, we must represent our data bits as individual matrices. Since we are using (7, 4) Hamming code, we will need four matrices to present each data bit as seen in Figure 4.9.. Since our data consists of 4 bits, we need matrices of dimensions 4×1. Each location of a 1 in the matrices represents the data bit position we want to represent.

$$\mathbf{D_1} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \mathbf{D_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad \mathbf{D_3} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \qquad \mathbf{D_4} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Figure 4.9. Matrix Representation of Data Bits

Using these matrix representations, we can now represent each parity bit as an individual matrix to construct the generator matrix. Instead of using the XOR operation, we can replace it with modulo-2 addition and recompute this equation set 4.2 with the data bit matrices to produce the equations in Figure 4.10..

Using the matrices in Figure 4.9. and linear equations set 4.10., the generator ma-

$$\mathbf{P_1} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \qquad \mathbf{P_2} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \qquad \mathbf{P_3} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Figure 4.10. Matrix Representation of Parity Bits

$$\mathbf{G} = \begin{array}{c} \begin{matrix} P_1 & P_2 & D_1 & P_3 & D_2 & D_3 & D_4 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

(a) Complete Generator Matrix (Non-Systematic)

$$\mathbf{G} = \begin{array}{c} \begin{matrix} D_1 & D_2 & D_3 & D_4 & P_1 & P_2 & P_3 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

(b) Complete Generator Matrix (Systematic)

$$\mathbf{g} = \begin{array}{c} \begin{matrix} P_1 & P_2 & P_3 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

(c) Sliced Generator Matrix

Figure 4.11. Generator Matrices

trix G can be represented as seen in Figure 4.11.a. This matrix can be used to generate a code word from any 4-bit data via the equation $c = DG$–where c is the code word, D is the 4-bit data, and G is the generator matrix. Given that columns D_1, D_2, D_3, and $D4$ only contain a single 1, we know that the bit values of the code word at these positions will be identical to the value of the data bits. Therefore, as opposed to computing these values via matrix operations, we can simply set these bits in the code word to be constant. With this in mind, we dropped these columns from the generator matrix, as seen in Figure 4.11.c reducing its dimensionality and computation needed with the generator matrix.

$$\mathbf{H} = \begin{array}{c} \begin{matrix} P_1 & P_2 & D_1 & P_3 & D_2 & D_3 & D_4 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \end{array} \qquad \mathbf{H} = \begin{array}{c} \begin{matrix} D_1 & D_2 & D_3 & D_4 & P_1 & P_2 & P_3 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

(a) Parity Check Matrix (Non-Systematic) (b) Parity Check Matrix (Systematic)

$$\mathbf{p^T} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

(c) Transposed Parity Bits Submatrix

Figure 4.12. Relevant Matrices for Constructing Parity Matrix

Hamming Decoding

Working in conjunction with the generator matrix, the parity check matrix determines if a code word contains a single error and, if so, allows for the bit to be corrected. The parity check matrix can easily be constructed from the generator matrix. This involves writing the generator matrix in systematic form, i.e. the data bits and parity bit are separated, as seen in Figure 4.11.b. Once in this form, we can extract the submatrix p of parity bits–this matrix is equivalent to that in Figure 4.11.c–and transpose it to result in the matrix p^T seen in Figure 4.12.c. Similar to the generator matrix, we concatenate an identity matrix with a column dimension equal to the number of parity bits to result in the systematic parity check matrix H, as seen in Figure 4.12.b. Assuming we used the non-systematic matrix to encode the weights, we must also ensure that the parity matrix H is also in this form. To convert the matrix back to non-systematic form, simply perform elementary matrix operations on the systematic form to result in the matrix depicted in Figure 4.12.a.

Through the equation $s = rH^T$–where r equals the data d plus any error e

introduced–we can calculate the syndrome s for any code word encoded from matrix G. The syndrome is the calculated value used to detect the presence and location of errors in a received code word. A list of syndromes for matrix H can be seen in Table 4.9.. As previously mentioned, placing parity bits at locations that are the power of 2 leads to a more efficient means of error correction. Given that we adhered to this statement, the generated syndromes correspond numerically to the location of the error (assuming the right-most bit is the most significant). Had we not placed our parity bits in this specific configuration, the syndromes would still give the location of the error, but they would not be equal to the bit positions.

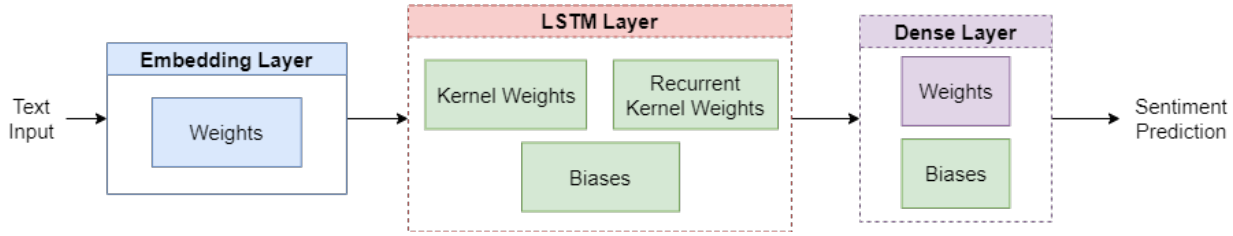
Table 4.9. Syndromes of Parity Matrix H

Syndrome $\mathbf{s = rH^T}$	Error Pattern
000	0000000
001	0001000
010	0100000
011	0000010
100	1000000
101	0000100
110	0010000
111	0000001

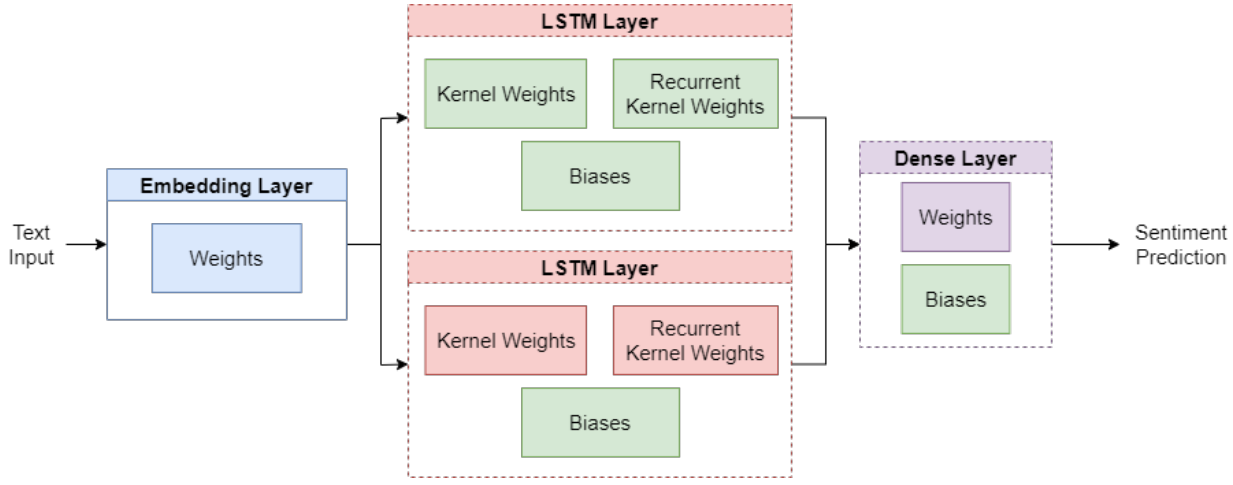
Note that if multiple errors occur, the syndrome table will not hold true, as this method of mitigation only works for a single-bit error in a set of four bits. Therefore, we can protect a total of 2 bit-flip errors, assuming they do not occur in the same set of 4 bits.

4.5.2. Fault Mitigation Implementation

To implement (7,4) Hamming code into our models, we first created encoding and decoding functions, as seen in the Appendix in Figures E.1. and E, specifically designed for the 8-bit exponent of FP32 numbers. We chose to only encode this section of bits, as it is the most susceptible to SEUs. Since the addition of these functions to the model invokes extra computation, it is important to only harden the exponent bit field of weights and biases that are susceptible to SEUs. For instance, we noticed that the embedding layer, in our case, was inherently resilient, even with exponent bit field SEUs. Therefore, implementing hamming code on these weights would introduce unnecessary computation.



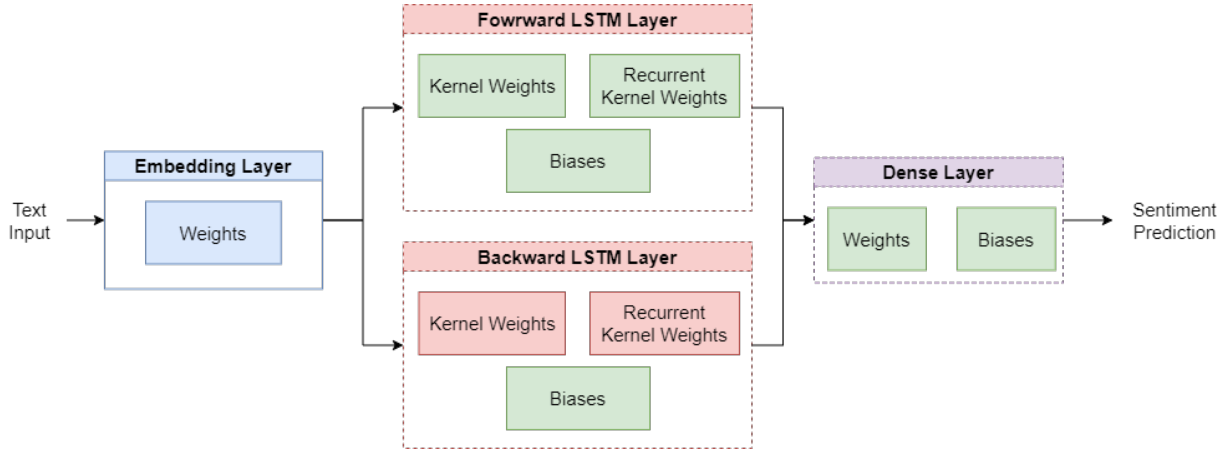
(a) LSTM Layer



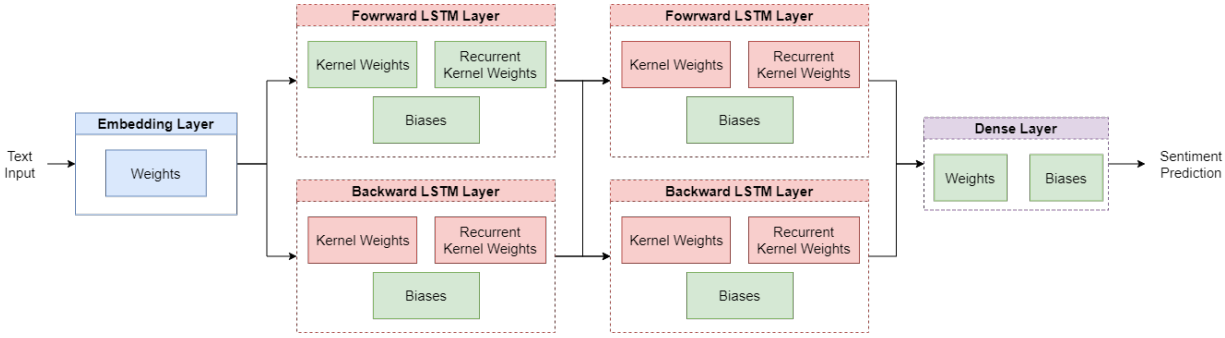
(b) Stacked LSTM Layers

Figure 4.13. LSTM Model Architectures Used for Sentiment Analysis

Since the initial accuracy of every model was approximately 86.5%, we selected sets



(c) Bidirectional LSTM Layer

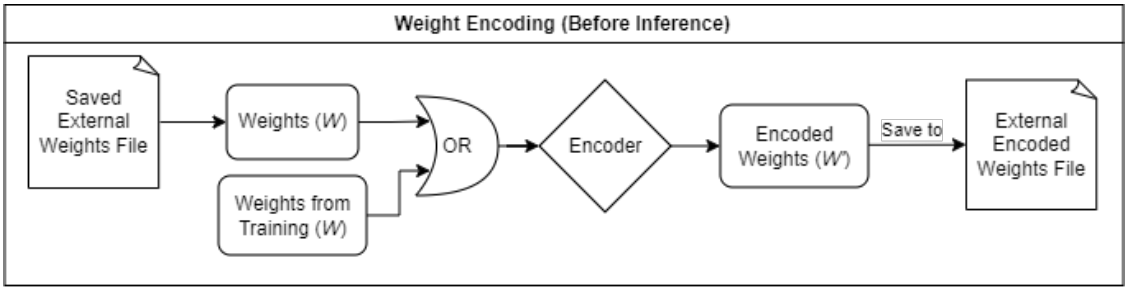


(d) Stacked Bidirectional LSTM Layers

Figure 4.13. (cont'd) Modified LSTM Model Architectures with Hamming Code

of weights and biases that produced an average accuracy of less than 80% when injected with 20 SEUs in the exponent bit field (See Figures 4.5. - 4.8.). We selected this specific testing case as it represented the worst-case scenario. The layers that we chose to harden for each network can be seen highlighted in green within Figure 4.13..

Table 4.10. Process of Encoding Weights of LSTM Network H



To implement the encoding, simply load the original weights from a saved weights file into the model or retrieve the weights directly after training the model, select which sets of weights to encode, pass them to the encoder function, and save the resulting weights back into the model and, then, into a weights file. This will allow us to possess a weights file that contains Hamming-encoded weights that can be loaded at any time by the LSTM model. This process can be seen more clearly in Figure 4.10..

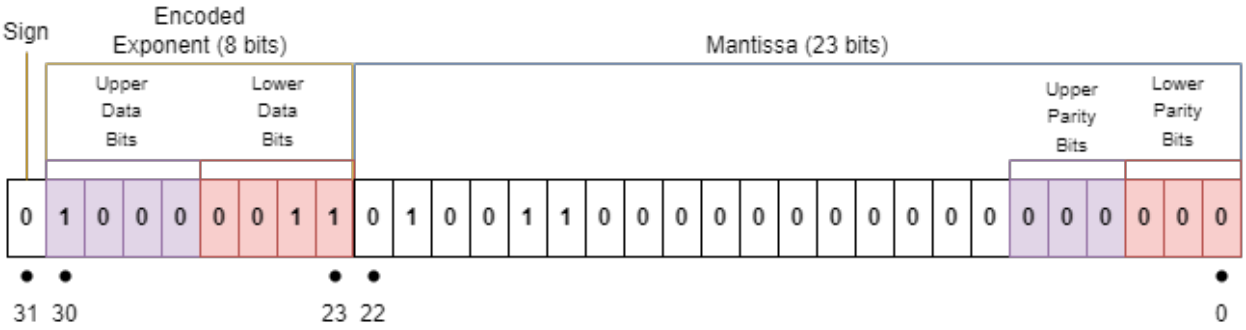


Figure 4.14. Arrangement of Data and Parity Bits for Hamming Encoding

As previously mentioned, we chose to encode the exponent bit field of susceptible weights and biases, as SEUs in this section of bits lead to SDC consistently. Since we are encoding two separate sets of 4-bit data with (7, 4) Hamming code, we will have a total of 6 parity bits. In other words, we will have an additional 6 bits for every encoded exponent bit field of a weight. Given that we are using FP32 numbers, we cannot expand the number of bits of these data types. Doing so would require us to use FP64, which would potentially lead to even more issues, as a wider range of values and bits would be available for SDC. Since we already proved that the mantissa was the least affected by changes in the bits, we chose to store each set of 3 parity bits in the least significant bits of each FP32 number, as seen in Figure 4.14.. This systematic arrangement of bits for encoding is handled by the encoder function, as seen in E.1.. Additionally, as shown in Table 4.11.,

this change in the least significant bits of weights and biases of the selected sets had little to no change in each model's accuracy.

Table 4.11. Accuracies of the Original and Modified LSTM Networks

Model	Original Accuracy (%)	New Accuracy (%)
LSTM	87.09999918937683	87.09999918937683
Stacked LSTM	86.72800064086914	86.72800064086914
Bidirectional LSTM	86.88799738883972	86.88799738883972
Stacked Bidirectional LSTM	87.81599998474121	87.81200051307678

Since the models were not trained on encoded weights, we must decode them to their original values. To do so, we modified the selected layers for hardening and wrapped instances of memory accesses to weights and biases with the "K.in_train_phase()" function. This specific function allows layers to behave differently during training and inferencing. This behavior is important to ensure that our method works for both untrained and trained models. Its functionality is similar to an if-else statement where the condition is determined by the model's deployment state. For the training state, we had the model utilize unencoded weights, allowing it to develop trained values for prediction. Assuming that we encode the weights after this training, the weights passed to the network will not be set to their trained values. Therefore, for the inference state, we have the model call our decoding function (hamming_decode_weights) on the passed encoded weights to restore them to their original, trained values. The changes and differences in the code can be seen in Figure 4.15..

In the event that 2 or fewer SEUs occur in the exponent bit field of the encoded weights during inference–assume that pairs of SEUs do not occur in the same 4-bit halves– our decoding function will restore the originally trained values to be used within the

```
z = backend.dot(inputs, self.kernel)

z += backend.dot(h_tm1, self.recurrent_kernel)

if self.use_bias:
z = backend.bias_add(z, self.bias)

```

(a) LSTM Layer Performing Weight & Bias Operations in without Decoders

```
z = backend.dot(inputs,
            K.in_train_phase(self.kernel,
            self.hamming_decode_weights(self.kernel))
            )

z += backend.dot(h_tm1,
             K.in_train_phase(self.recurrent_kernel,
             self.hamming_decode_weights(self.recurrent_kernel))
            )

if self.use_bias:
z = backend.bias_add(z,
                     K.in_train_phase(self.bias,
                     self.hamming_decode_weights(self.bias))
                    )

```

(b) LSTM Layer Performing Weight & Bias Operations in with Decoders

Figure 4.15. Code Segment Example of LSTM Layer With and Without Hamming Encoded Weights

model. However, if 3 or more SEUs occur in the model, the decoder function will not be able to detect or correct any errors. Therefore, the error will propagate through the network. These cases of decoding can be seen in Figure 4.16..

All matrix operations for this implementation were done through Keras and Tensorflow APIs to ensure that any Keras/Tensorflow-based model could utilize our mitigation methods without potential dependency issues. Additionally, this implementation does not require any retraining of the models. For untrained models, users must simply implement the decoder in the layers they want to harden, wrap selected weights and biases with the function, train the model, and encode the weights, saving it to a weights file. Simi-

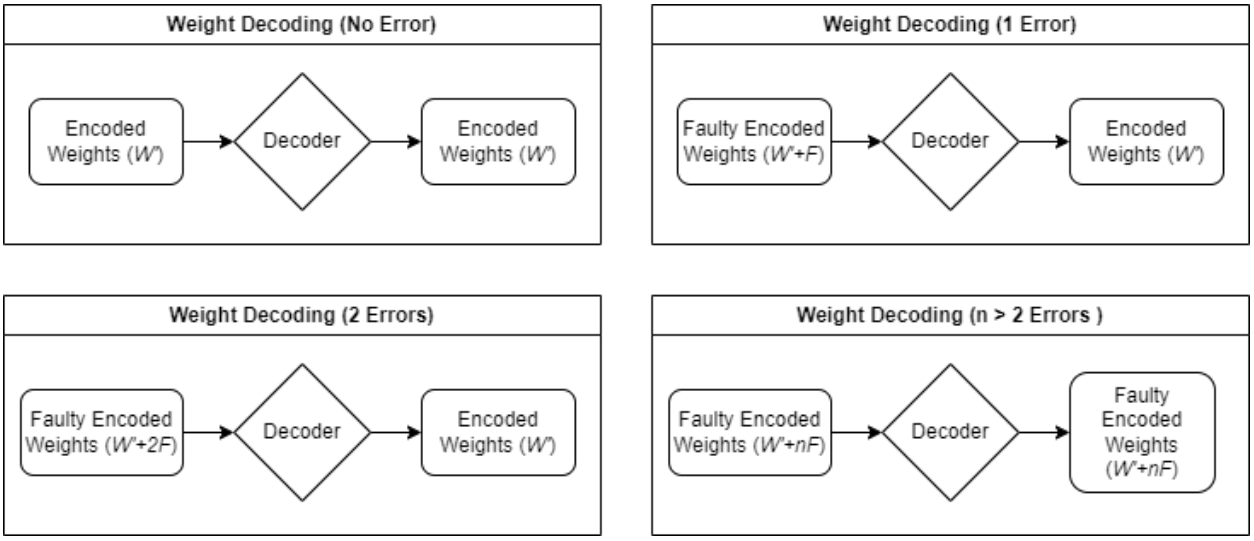


Figure 4.16. Decoder Behavior Under Various Number of SEUs (Assuming Pairs of SEUs Do Not Occur in the Same 4-Bit Halves of the Exponent Section)

larly, for trained models, users will proceed through the same steps, except will not need to train the model. In both cases, as shown in Figure 4.17., the models can then load the encoded weights file and be deployed with increased resilience.

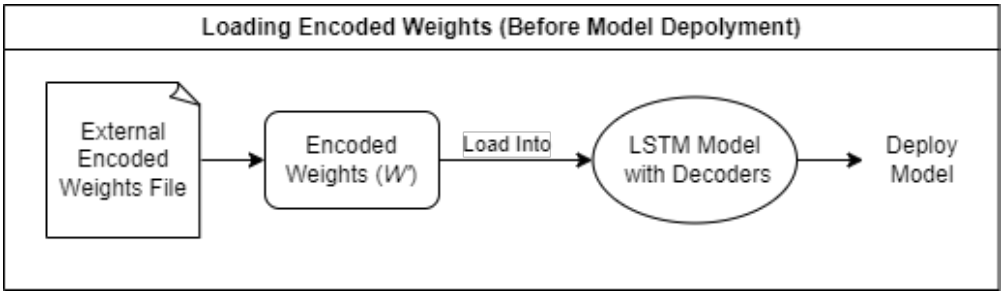


Figure 4.17. Process of Using Encoded Weights with LSTM Model

4.5.3. Mitigation Results & Analysis

To maintain consistency in testing, we maintained injecting SEUs in sets of 1, 10, and 20 for 100 iterations for each layer. However, since we only hardened specific sets of weights and biases within the network, we only performed injection tests on the sets with our mitigation method implemented. Furthermore, given that we only hardened the expo-

nent bit field of the weights and biases, we performed injection into only the exponent for our bit field tests. In addition to this set of injections, we also continued to inject all the bits to verify that layers that have our decoding method do not affect the overall performance of the model even when they are not the ones experiencing errors.

Table 4.12. Matricies of the Average Accuracies of Single LSTM Layer Model's Susceptible Weight/Bias Sets After 1, 10, and 20 SEUs Over 100 Iterations

			Injected Layer & Weight/Bias Set			
			LSTM 1			Dense
			KW	RKW	B	B
Injected Bit Field & Number of Injections Performed per Iteration	All Bits	1	87.10	87.10	86.97	87.10
		10	87.10	86.99	86.29	74.48
		20	86.97	86.85	84.00	63.34
	Exponent Bits	1	87.10	87.10	87.10	87.10
		10	87.10	87.08	85.20	N/A
		20	86.82	86.40	77.27	N/A

Focusing on the single LSTM layer model's results seen in Table 4.12., we noticed improvements in the accuracy of up to 20% in the LSTM layer. More specifically, for almost all exponent injection tests, the accuracy did not fall below 85%. The only exception to this fact was in the case of 20 SEUs occurring in the biases of the LSTM layer. This result is most likely due to the high fault percentage of 50% for the biases in the LSTM

layer caused by 20 SEUs. Similarly, the injections into all the bits of the weights and biases only produced two instances of the accuracy being less than 82%. Both of these instances occurred when multiple bits of the dense layer's bias were flipped. Given that the bias is only a single number, any injection into this weight results in a fault percentage of 100%. Therefore, hardening such a weight with our mitigation method may not be feasible. Overall, these results show a massive improvement over the counterpart results of the original first model which was littered with accuracies lower than 75%. Additionally, we see small general improvements in accuracy for almost all tests.

Table 4.13. Matricies of the Average Accuracies of Stacked LSTM Layers Model's Susceptible Weight/Bias Sets After 1, 10, and 20 SEUs Over 100 Iterations

			Injected Layer & Weight/Bias Set				
			LSTM 1			LSTM 2	Dense
			KW	RKW	B	B	B
Injected Bit Field & Number of Injections Performed per Iteration	All Bits	1	86.72	86.70	86.71	86.72	86.72
		10	86.72	86.63	85.94	86.72	74.86
		20	86.71	86.58	83.86	86.67	66.22
	Exponent Bits	1	86.72	86.72	86.68	86.72	86.72
		10	86.56	86.70	83.71	86.66	N/A
		20	86.70	85.93	77.14	86.47	N/A

The results of the modified stacked LSTM model are almost the same as the previous model's. However, one noticeable improvement is the results of the second LSTM

layer being injected with SEUs. In the original second model, the second LSTM layer experienced a large decrease in performance in its biases. With mitigation, the rate of SDC is almost completely eliminated from this bias set, as the accuracy of it is maintained at a steady 86% for all injection tests. Aside from this difference, the results of this model are consistent with the previous model's instances of poor accuracy and overall improved performance over the original model.

Table 4.14. Matrices of the Average Accuracies of Bidirectional LSTM Layer Model's Susceptible Weight/Bias Sets After 1, 10, and 20 SEUs Over 100 Iterations

			Injected Layer & Weight/Bias Set					
			Forward LSTM 1			Backward LSTM 1	Dense	
			KW	RKW	B	B	KW	B
Injected Bit Field & Number of Injections Performed per Iteration	All Bits	1	86.88	86.87	86.68	86.88	86.88	86.88
		10	86.88	86.30	85.72	86.88	85.56	73.20
		20	86.88	86.54	82.47	86.49	83.29	64.52
	Exponent Bits	1	86.88	86.88	86.88	86.89	86.88	86.88
		10	86.60	86.76	83.30	86.91	82.91	N/A
		20	86.87	86.39	75.17	86.93	72.50	N/A

Just as with the other models, the accuracies for the bidirectional LSTM model in Table 4.12. were generally higher for almost every test. As mentioned before, the dense layer's bias continues to be a weak point due to the minimum number of biases present

in the layer. In line with the previous model's results, the accuracy of a single SEU being introduced in the exponent bit field has been improved and has risen near the highest attainable accuracy. However, multiple injections in any of the bias's bits still reduce the accuracy tremendously. However, the hardening of the dense layer's kernel weights resulted in significant improvements over the original model. Specifically, for 10 SEUs in the exponent, the accuracy has risen from 70.15% to 85.56%. Under 20 SEUs, the kernel weights also experience improvement, but the accuracy is still relatively poor. For the remaining two layers, their improvements in accuracy and areas of low accuracy are identical. In other words, the first LSTM layer's results of the stacked LSTM model correspond to the forward LSTM's results. Likewise, the results of the second LSTM layer's biases of the stacked LSTM model correspond to the first backward LSTM layer's. Given that these two layers are so architecturally similar, the parallelism of their results is to be expected. Based on this observation, the results of this model continue to show the improved resilience of LSTM networks when introducing our mitigation method.

Continuing the pattern of improvement, the fourth LSTM model showed the best performance in maintaining accuracies of over 85.64 and 84.5 for tests with all bits and the exponent bit field, respectively (See Figure 4.13.). The only exception to this fact is the areas of susceptibility previously mentioned in the other networks. Looking at the biases of the various layers we encoded, we were able to maintain accuracies higher than 86% for all tests. This is a huge improvement over the original stacked bidirectional LSTM model. All of the results for each individual injection test with corresponding models can be more clearly seen in Appendix Figures D.1. - D.4..

Looking at the areas of poor accuracy for all models, the biases of the subsequent

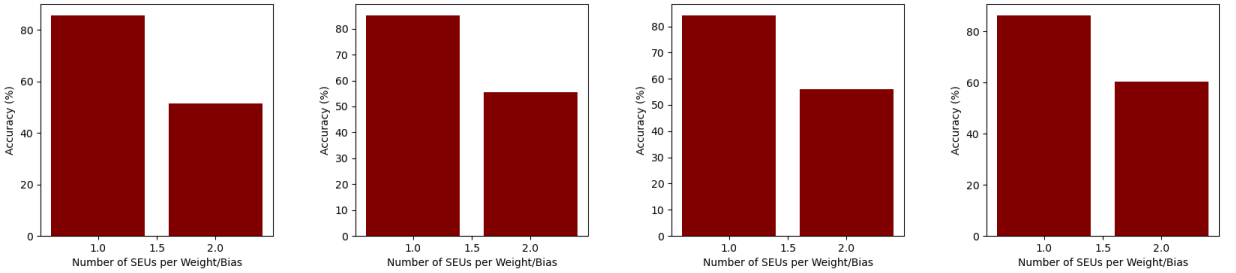
Table 4.15. Matricies of the Average Accuracies of Stacked Bidirectional LSTM Layer Model's Susceptible Weight/Bias Sets After 1, 10, and 20 SEUs Over 100 Iterations

			Injected Layer & Weight/Bias Set							
			Forward LSTM 1			Backward LSTM 1	Forward LSTM 2	Backward LSTM 2	Dense	
			KW	RKW	B	B	B	B	KW	B
Injected Bit Field & Number of Injections Performed per Iteration	All Bits	1	87.81	87.81	87.77	87.81	87.81	87.81	87.81	87.81
		10	87.81	87.69	86.26	87.80	87.43	87.81	87.79	77.22
		20	87.73	87.22	85.64	87.32	87.79	86.31	85.68	64.75
	Exponent Bits	1	87.81	87.81	87.78	87.81	87.81	87.81	87.81	87.81
		10	87.78	87.51	84.52	87.37	87.81	87.81	85.31	N/A
		20	87.64	87.44	77.34	86.49	87.64	87.82	79.30	N/A

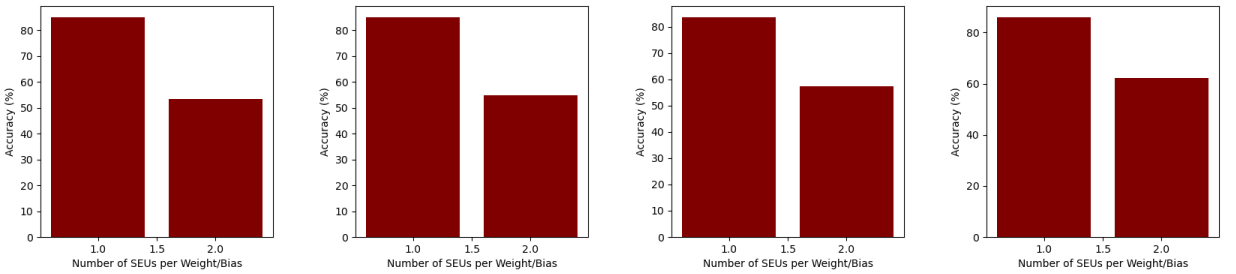
layer of the embedding layer and the dense layer consistently gain the lowest accuracies with increasing SEUs. Given that this pattern has repeated throughout all models, we can reason that our mitigation method is not effective on sets of weights and biases with small numbers of elements. Therefore, weight and bias set with smaller weights sound not use this method, as it will introduce unnecessary costs.

For all the models, this negative behavior is caused by the increase in the probability of multiple SEUs occurring in either the upper or lower half of the exponent bit field. In fact, based on the multi-SEU injections performed on the modified networks, we were able to see the impact of multiple SEUs being introduced in either the upper or lower half of the exponent bit-field (See Figure 4.18.).

Despite this poor performance in the worst case, we plotted the frequency at which these multiple SEUs occurred in the upper and lower halves of the exponent bit field. As



(a) LSTM Layer (Upper Bits) (b) Stacked LSTM Layers (Upper Bits) (c) Bidirectional LSTM Layer (Upper Bits) (d) Stacked Bidirectional LSTM Layers (Upper Bits)



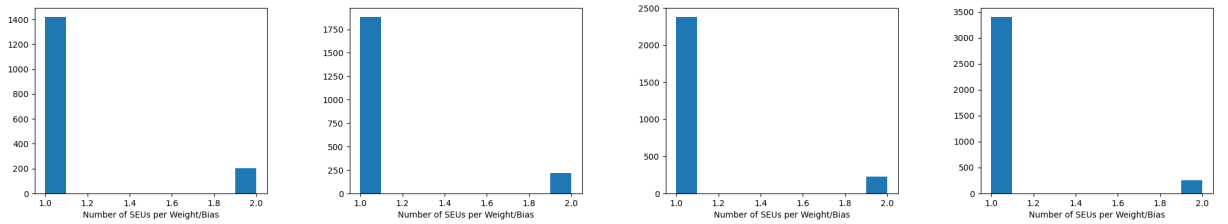
(e) LSTM Layer (Lower Bits) (f) Stacked LSTM Layers (Lower Bits) (g) Bidirectional LSTM Layer (Lower Bits) (h) Stacked Bidirectional LSTM Layers (Lower Bits)

Figure 4.18. Accuracy of Multiple SEUs Occurring within the Same Weights for the Upper and Lower Bits of the Exponent Bit Field (Based on Injections of 10 & 20 SEUs in Modified Models)

seen in Figure 4.19., the occurrence of two SEUs occurring in either half of the bit field was extremely small in comparison to the frequency of a single SEU. Additionally, there were no occurrences of more than two SEUs in a given half of the exponent bit field. This analysis shows the probability of the worst case occurring, i.e. more than 1 bit-flip per half of the exponent bit field, is extremely low.

As a final analysis of our model, we measured the execution time of the original model against our modified model. This experiment was conducted by having each model compute prediction for the Large Movie Dataset's test set (25000 samples) for 10 iterations. The times for each iteration were then averaged to give us the values shown in Table 4.16..

Given that we introduced additional computation into the model, we expected to

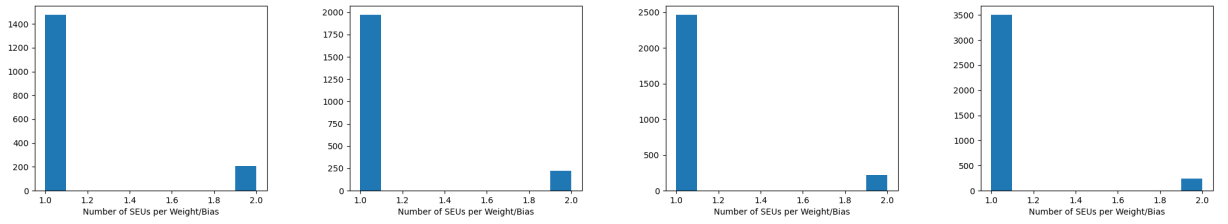


(a) LSTM Layer (Upper Bits)

(b) Stacked LSTM Layers (Upper Bits)

(c) Bidirectional LSTM Layer (Upper Bits)

(d) Stacked Bidirectional LSTM Layers (Upper Bits)



(e) LSTM Layer (Lower Bits)

(f) Stacked LSTM Layers (Lower Bits)

(g) Bidirectional LSTM Layer (Lower Bits)

(h) Stacked Bidirectional LSTM Layers (Lower Bits)

Figure 4.19. Frequency of Multiple SEUs Occurring within the Same Weights for the Upper and Lower Bits of the Exponent Bit Field (Based on Injections of 10 & 20 SEUs in Modified Models)

Table 4.16. Average Execution Times Over 10 Iterations of Original and Modified LSTM Models on Large Movie Dataset's Test Set (25000 Samples)

Model	Original Execution Time (sec)	New Execution Time (sec)
LSTM	4.902	15.824
Stacked LSTM	8.278	21.666
Bidirectional LSTM	8.304	26.292
Stacked Bidirectional LSTM	15.142	35.840

see an increase in the average execution time of the model. Based on the obtained results, we can observe a clear tradeoff between execution time and model resilience. Specifically, our modified models ran anywhere from 2 to 3 times the original execution time. Therefore, when deciding to harden specific weight sets or layers, it is important to consider the use case of the model and the number of elements being hardened, as each incurs a cost.

Chapter 5. Limitations & Future Work

When we initially began this study, we had intended to compare LSTMs to more recent transformer networks such as Bidirectional Encoder Representations from Transformers (BERT)[7]. However, due to hardware limitations, experiments with these architectures could not be performed. However, given that we chose sentiment analysis as the foundation of our models, this study can easily be extended into transformers, as they can also solve perform tasks. In fact, we hope that future research can investigate this mitigation method in the context of other types of models, data sets, and problems.

In addition, although our implementation was done in TensorFlow, we encourage others to implement this methodology in other deep neural network frameworks such as PyTorch. Such an implementation could be compared to ours to measure the overall efficiency of each. Furthermore, future work in this area could lead to a more efficient means of implementing hamming code, leading to lower computation overhead. In fact, a study into determining exact weights and biases within each set could prove to be a solution to massively reducing the overhead of this Hamming mitigation method.

As previously mentioned, this method of mitigation is intended to be used to harden weights and biases of LSTM networks. However, there still exists the threat of faults occurring during computation in the arithmetic logic unit (ALU) or even the output layers; therefore, combining our method with another that accounts for faults in the ALU or output layers could prove to be a method to protect against a wider arrangement of faults. This would also remove the limitation of our method with regard to hardening the output layer, i.e. the dense layer in our case.

Chapter 6. Conclusion

By exploring the resilience of LSTM networks, we discovered that, like many other neural networks, LSTMs share similar susceptibilities to other neural networks due to the usage of FP32. Usage of this data type with neural network models leaves an excessive range for the weights and biases, leading to SEUs having major impacts on the performance of models. Additionally, we also saw that different layers and sets of weights and biases of LSTM networks can have varying susceptibilities. However, a common area of susceptibility was the exponent bit-field of FP32 numbers. Through these discoveries, we were able to use Hamming code with the weights and biases to reduce the rate of SDC within the neural networks, leading to increased accuracies under various testing conditions. Although our implementation of this method incurs a high overhead, we hope that future research can build upon the methodology and combine it with others to improve the overall resilience of neural networks.

Appendix A. Supplemental Figures for Fault Injection Process

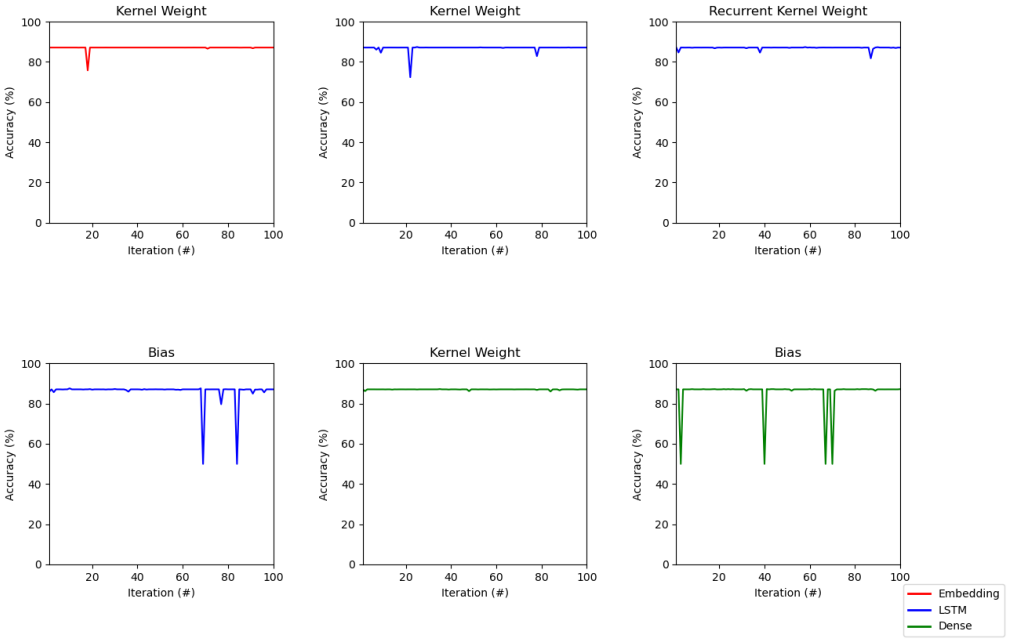
```
Load, Preprocess, and Split Dataset

Construct Model's Neural Network Architecture
Load Saved Model
Compile the Model

Set path to YAML config file
Set path to output data directory
Create a CSV file to store data

Get Number of Injections
Get Number of Random Samples
Set offset variable
Get Number of Samples in Test Set
Set Total Error to 0

Create empty array IND to hold samples classified correctly by model
Get number of (Random Samples + Offset) from test set based on the
   command line argument
Load the model's weights

for each Sample in (Random Samples + Offset):
    Get the test loss and accuracy with model
  if(test accuracy is 100%):
    Add to IND

Slice IND to length of (Number of Random Samples)

Set Timer to time fault injection
for each Injection in Number of Injections:
    Load the model's weights
    Call TensorFI 2 to perform fault injection
    Set Local Error to 0

    for each Sample in IND:
        Get the test loss and accuracy with model
      if(test accuracy is 0%):
        Add 1 to the Local Error

    Write (Local Error)/(Number of Injections) to CSV
    Add Local Error to Total Error

Write (Total Error)/((Number of Injections)*(Number of Samples)) to CSV
Write time taken for Fault Injection based on Timer to CSV
```

Figure A.1. Peudocode for Fault Injection Process

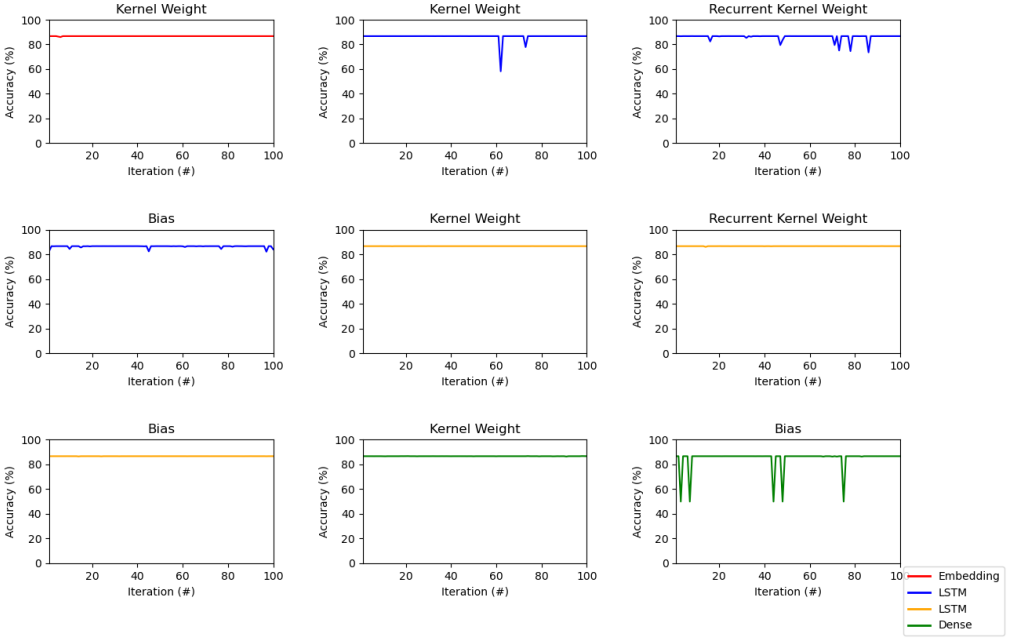
```
21/01/2023 19:29:58
----------------Starting fault injection in a random layer
    ----------------

Type of Fault: bit flips
Amount of Faults: 1

Original Layer [0]:
[[ 0.01407452 -0.02264381  0.04336489 ...  0.08614103  0.03903691
  -0.03838438]
 [ 0.04831325 -0.02108465 -0.00100701 ...  0.0746794   0.01041782
  -0.01459506]
 [-0.0508039   0.04833143  0.01402921 ... -0.0316338  -0.00736912
   0.03181344]
 ...
 [-0.08966255  0.0074457  -0.02848361 ...  0.01565933 -0.05324383
   0.00379305]
 [ 0.00273432 -0.04053009 -0.01886841 ...  0.09057098  0.03151694
  -0.0232784 ]
 [ 0.07806322  0.0476468   0.04622335 ... -0.08002724  0.07562602
  -0.03785311]]

Number of Elements in Layer: 75000
Layer Element Indicies (#s) to Inject: [2326]
(0) Bit Position in 32-Bit Element #2326 to be Flipped: 17
(0) Original #2326 Value: 0.06409169733524323
(0) Faulty #2326 Value:   0.06311513483524323

Faulty Layer [0]:

[[ 0.01407452 -0.02264381  0.04336489 ...  0.08614103  0.03903691
  -0.03838438]
 [ 0.04831325 -0.02108465 -0.00100701 ...  0.0746794   0.01041782
  -0.01459506]
 [-0.0508039   0.04833143  0.01402921 ... -0.0316338  -0.00736912
   0.03181344]
 ...
 [-0.08966255  0.0074457  -0.02848361 ...  0.01565933 -0.05324383
   0.00379305]
 [ 0.00273432 -0.04053009 -0.01886841 ...  0.09057098  0.03151694
  -0.0232784 ]
 [ 0.07806322  0.0476468   0.04622335 ... -0.08002724  0.07562602
  -0.03785311]]

Completed injections... exiting

```

Figure A.2. Example of Log File

Appendix B. Supplemental Results of Experiment 1 Injection Results

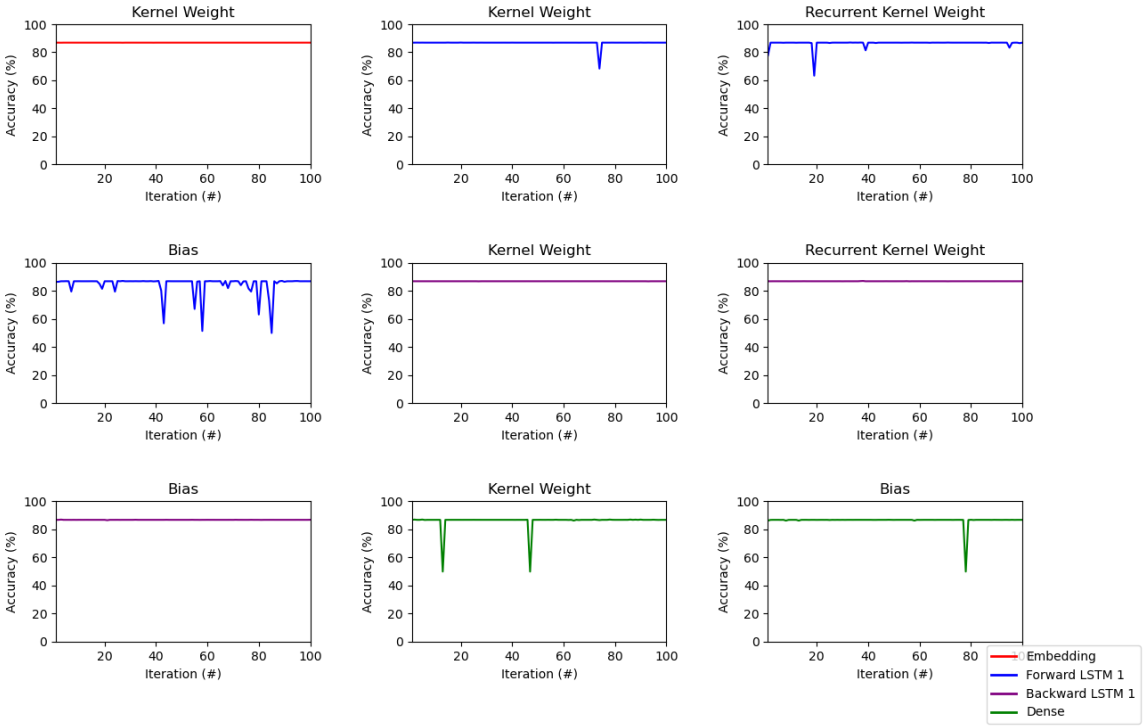


(a) LSTM Layer

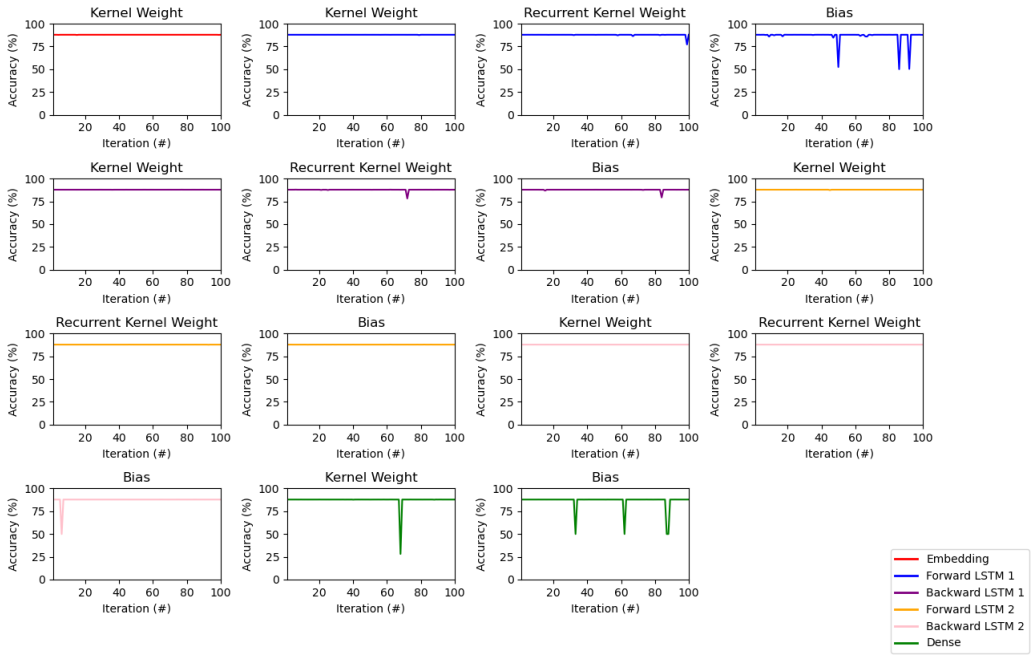


(b) Stacked LSTM Layers

Figure B.1. Original LSTM Models with 1 SEU in Each Set of Weights/Biases in Each Layer Over 100 Iterations

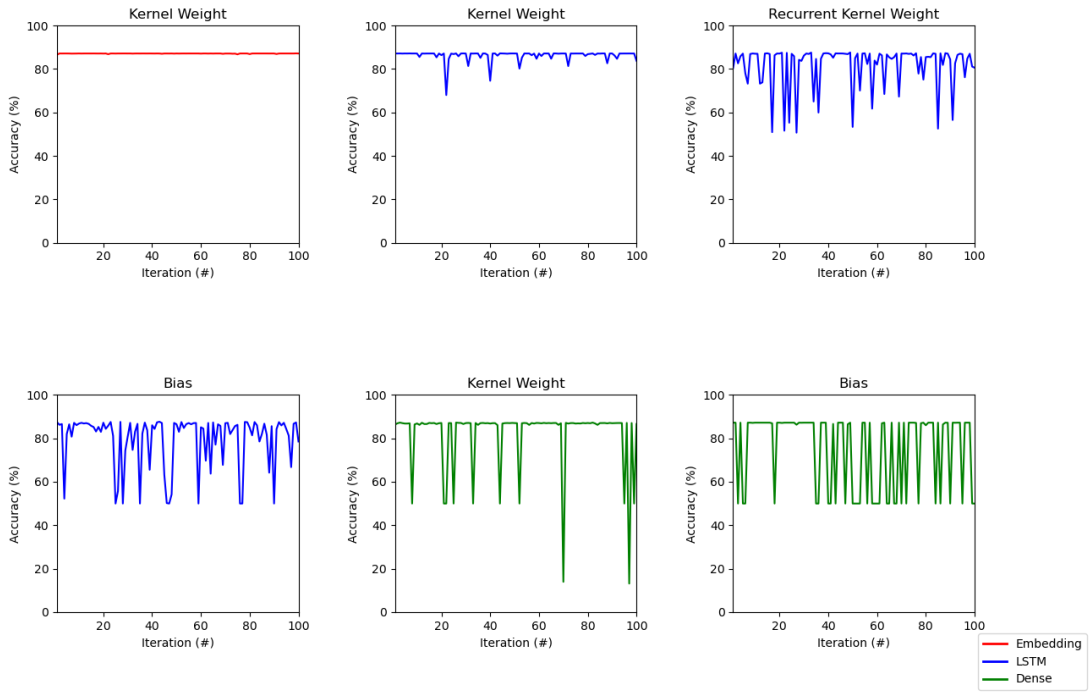


(c) Bidirectional LSTM Layer

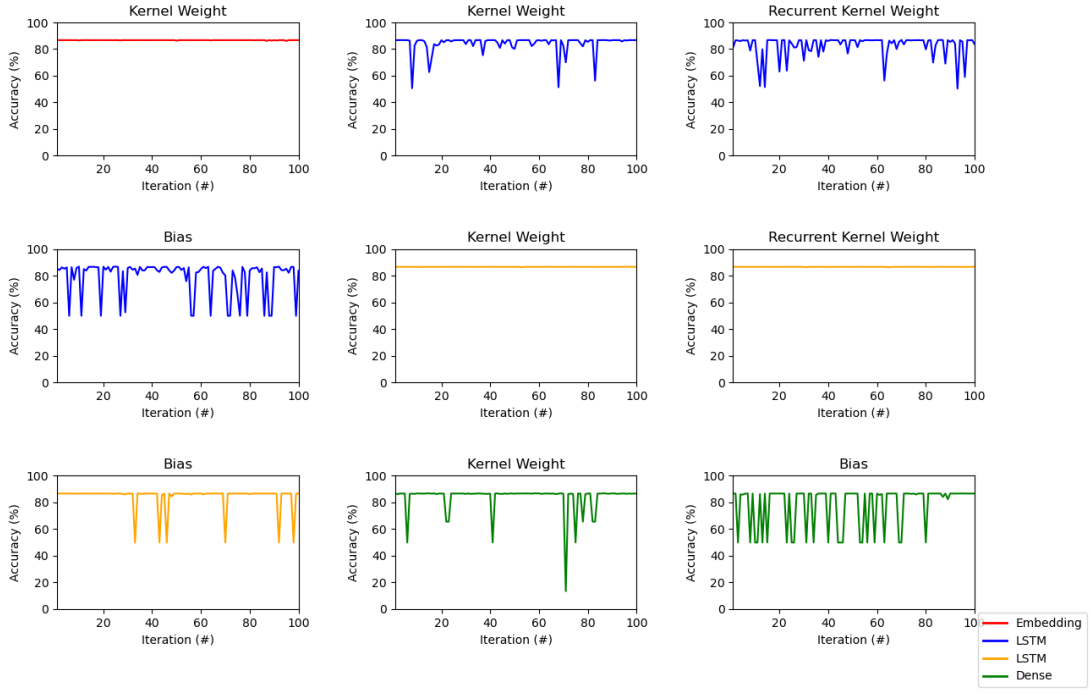


(d) Stacked Bidirectional LSTM Layers

Figure B.1. (cont'd) Original LSTM Models with 1 SEU in Each Set of Weights/Biases in Each Layer Over 100 Iterations

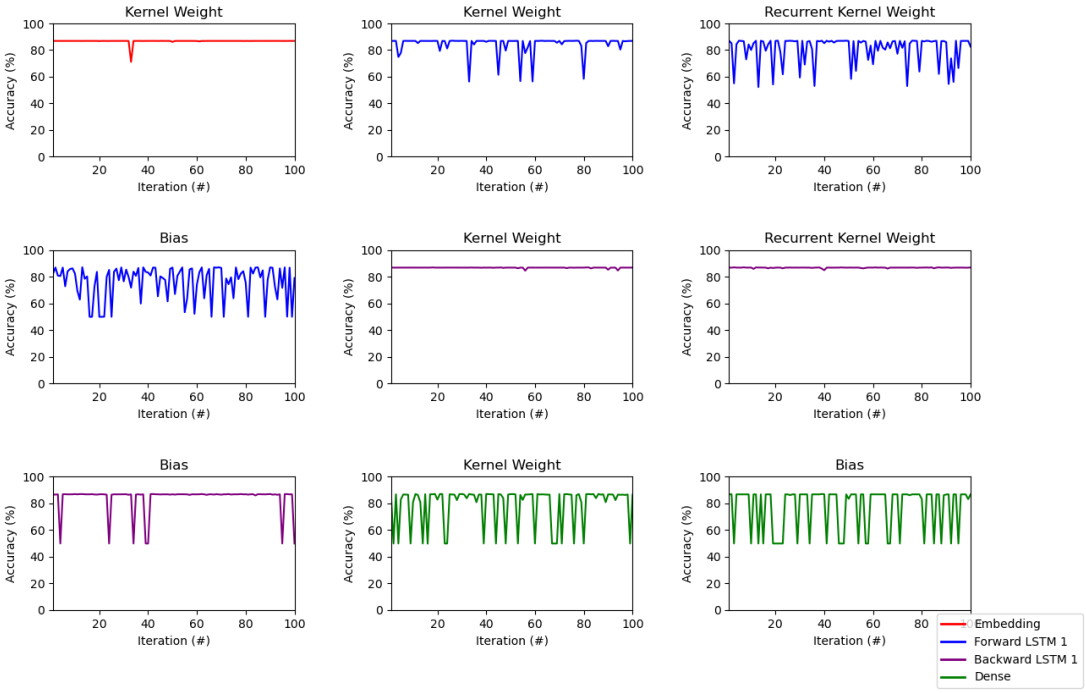


(a) LSTM Layer

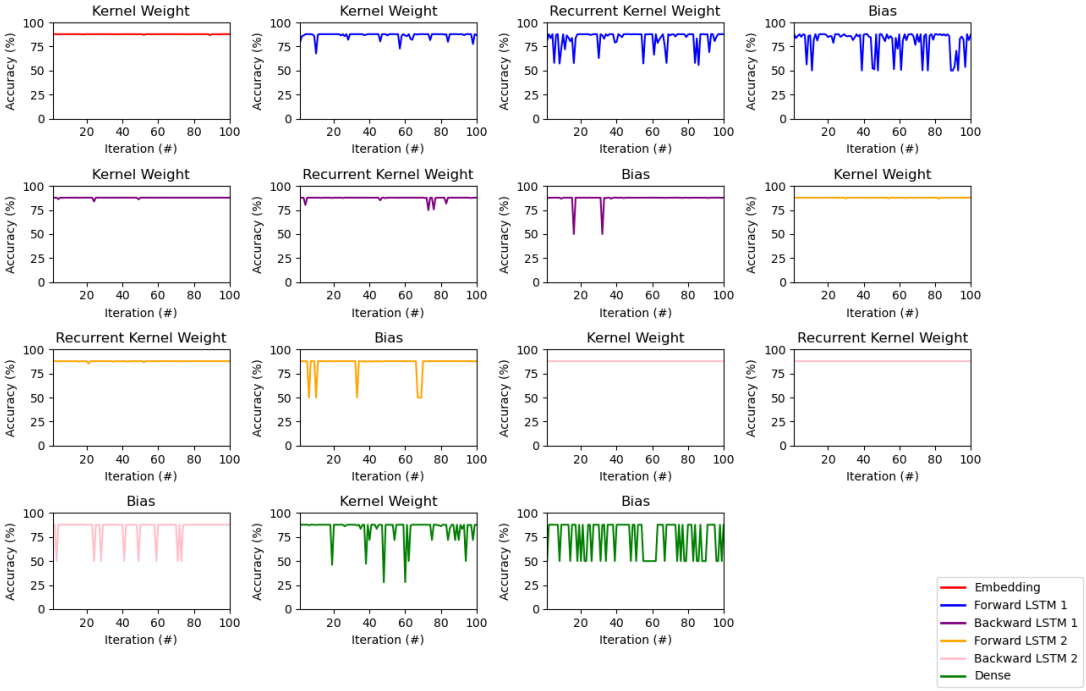


(b) Stacked LSTM Layers

Figure B.2. Original LSTM Models with 10 SEU in Each Set of Weights/Biases in Each Layer Over 100 Iterations

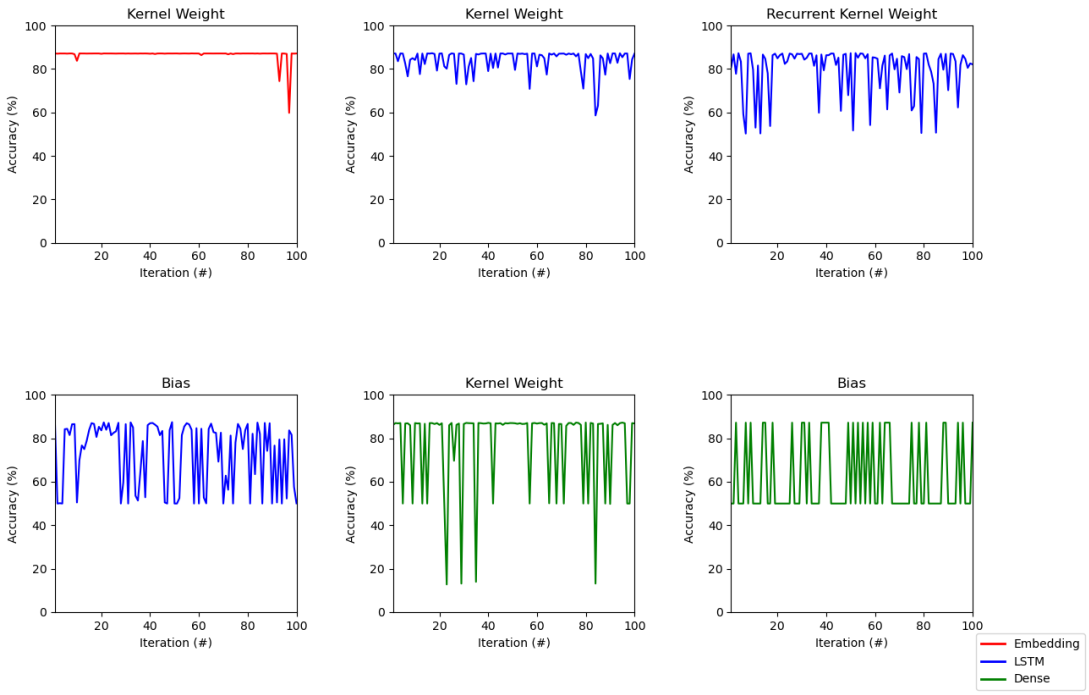


(c) Bidirectional LSTM Layer

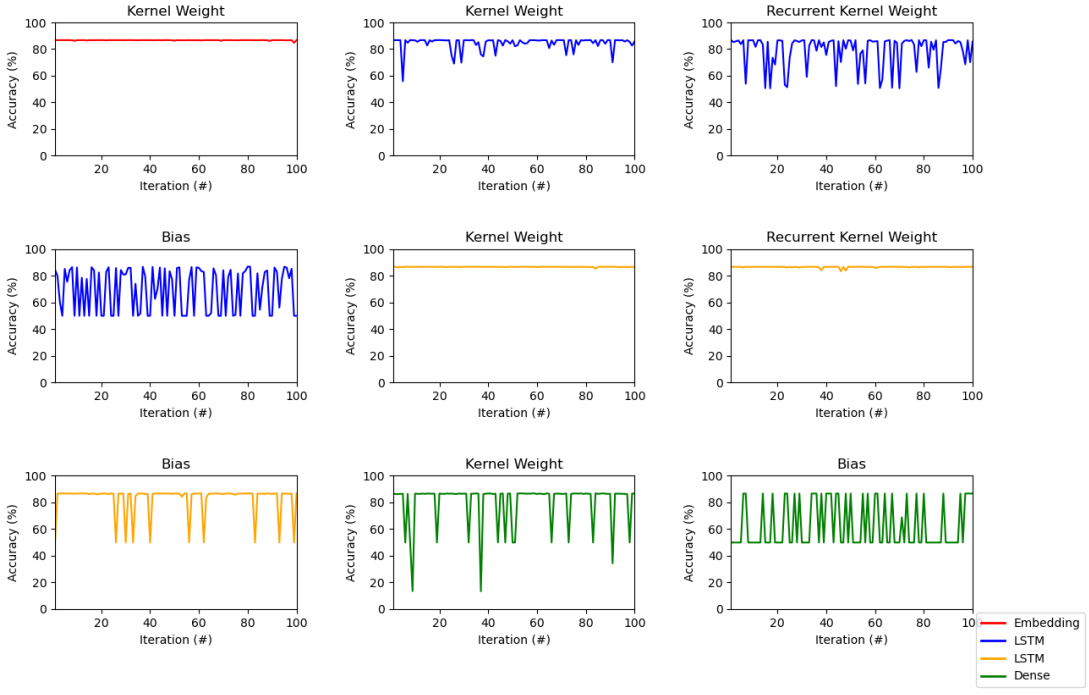


(d) Stacked Bidirectional LSTM Layers

Figure B.2. (cont'd) Original LSTM Models with 10 SEU in Each Set of Weights/Biases in Each Layer Over 100 Iterations

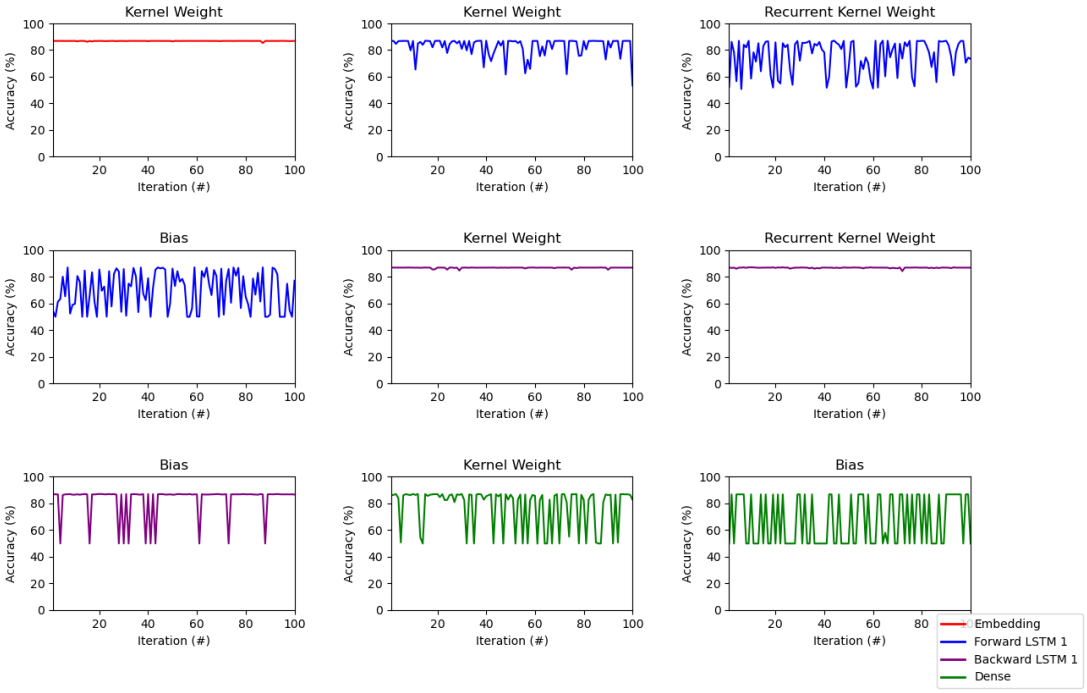


(a) LSTM Layer

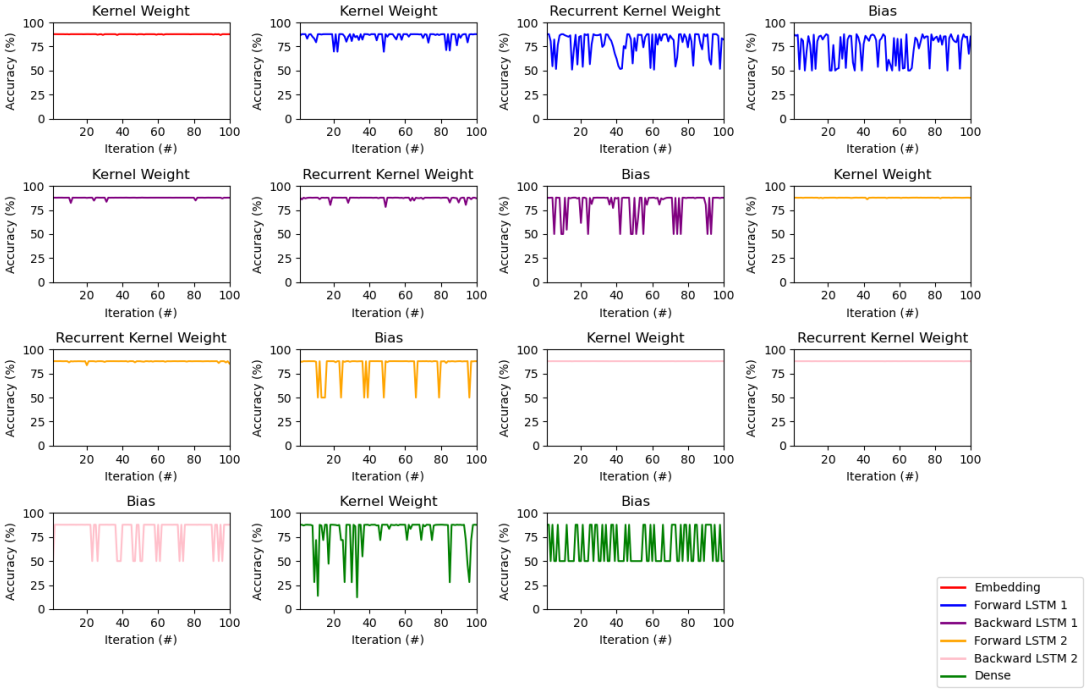


(b) Stacked LSTM Layers

Figure B.3. Original LSTM Models with 20 SEU in Each Set of Weights/Biases in Each Layer Over 100 Iterations

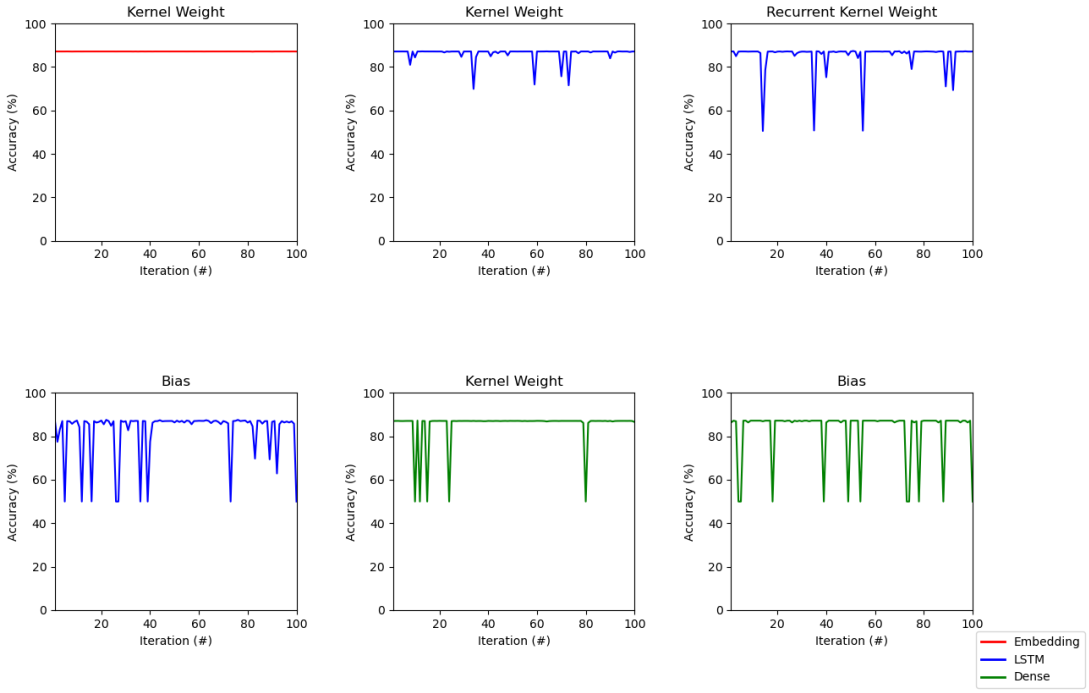


(c) Bidirectional LSTM Layer

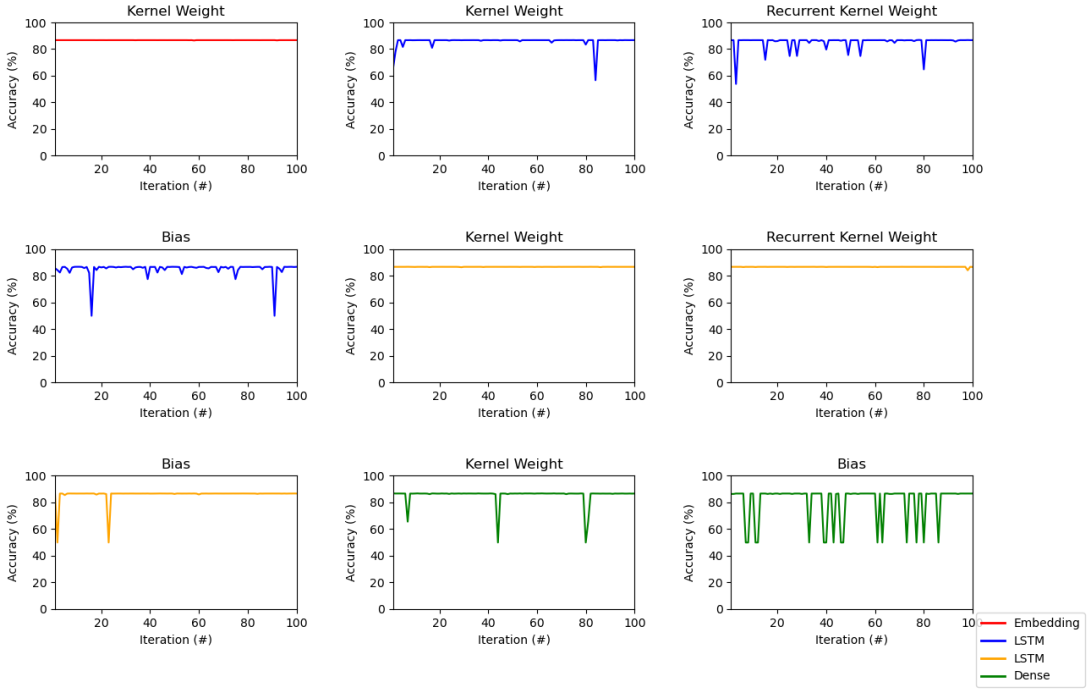


(d) Stacked Bidirectional LSTM Layers

Figure B.3. (cont'd) Original LSTM Models with 20 SEU in Each Set of Weights/Biases in Each Layer Over 100 Iterations

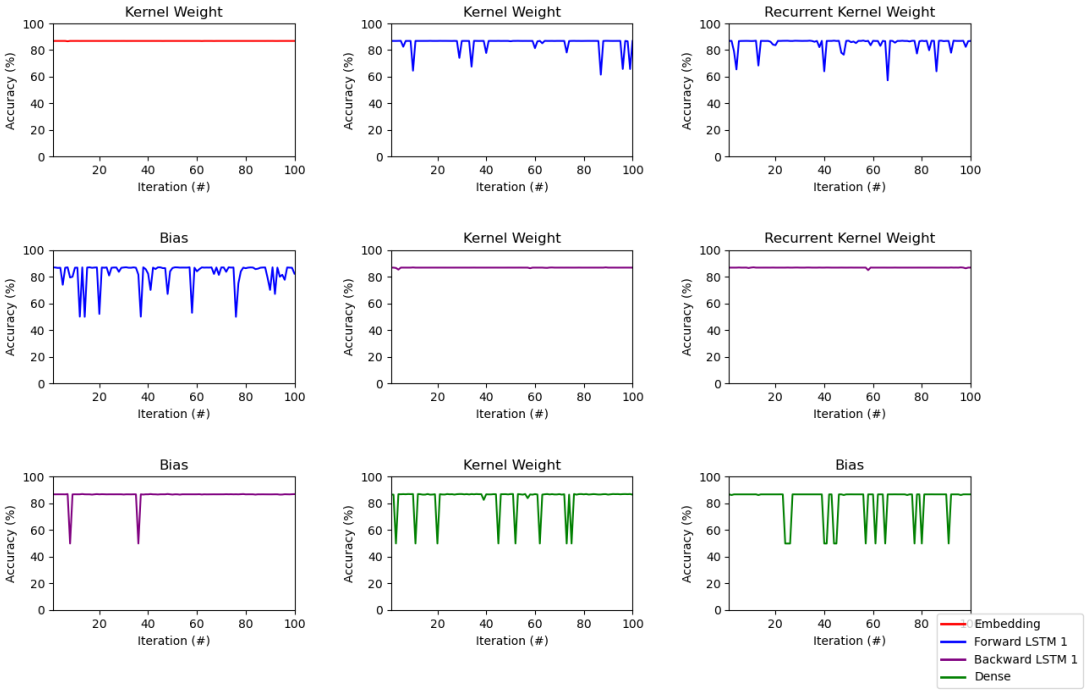


(a) LSTM Layer

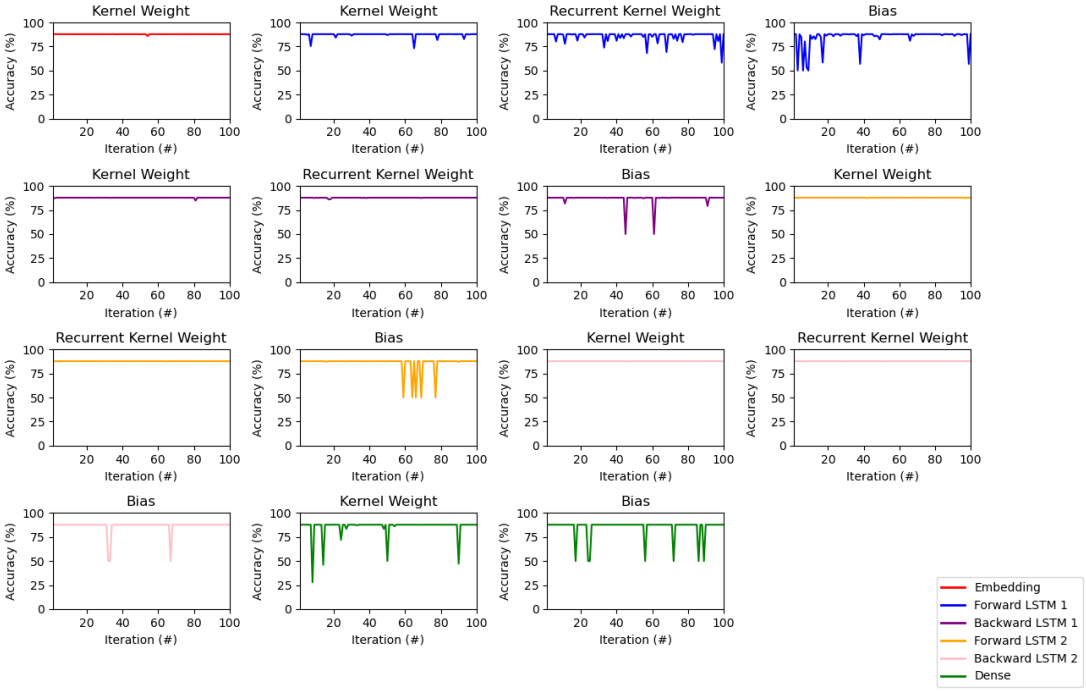


(b) Stacked LSTM Layers

Figure B.4. Original LSTM Models with 1 SEU in Exponent Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

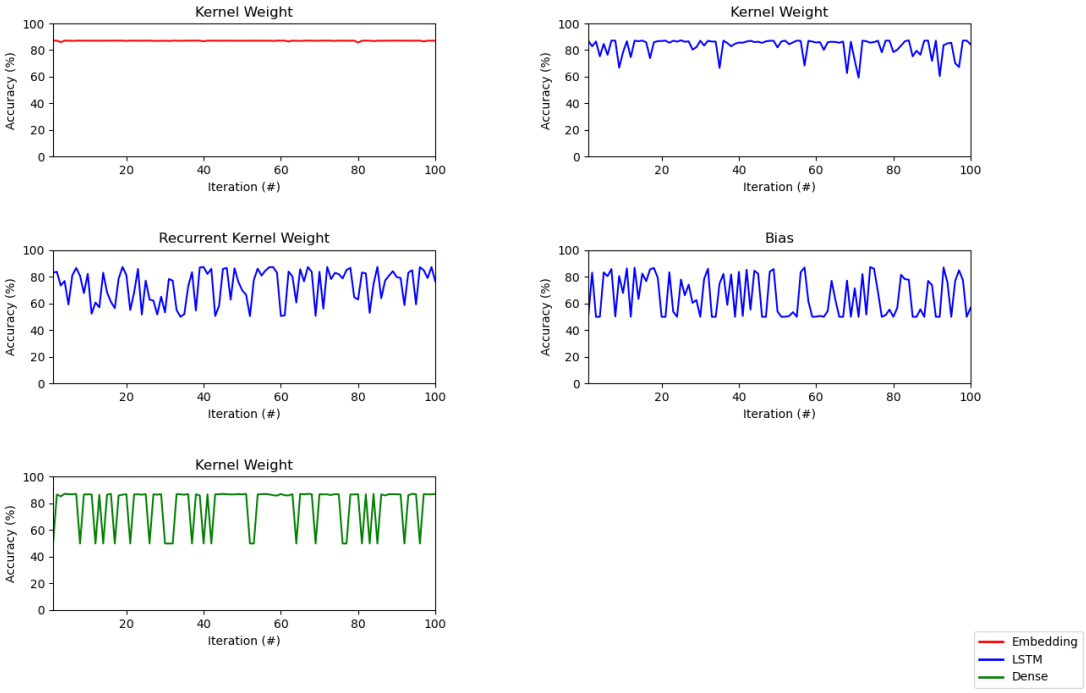


(c) Bidirectional LSTM Layer

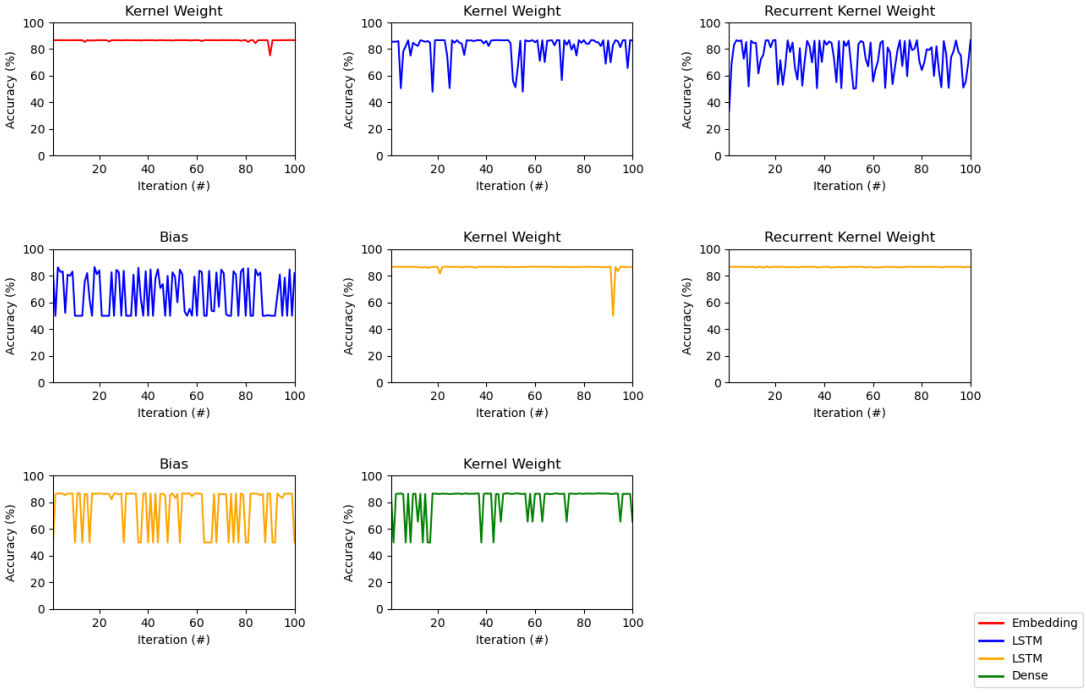


(d) Stacked Bidirectional LSTM Layers

Figure B.4. (cont'd) Original LSTM Models with 1 SEU in Exponent Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

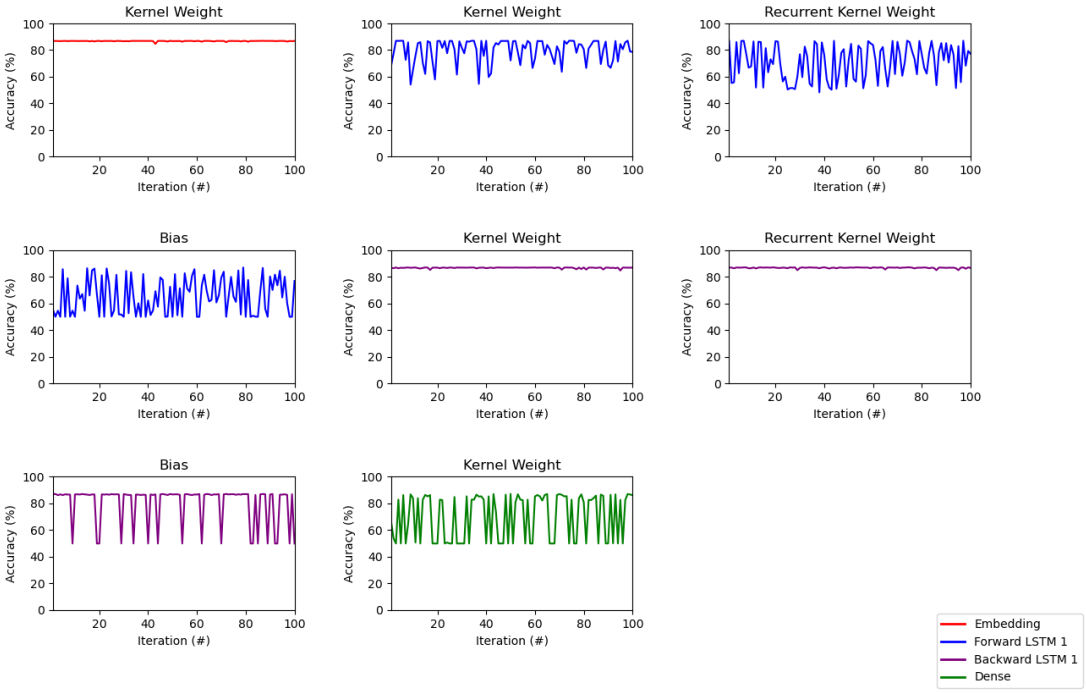


(a) LSTM Layer

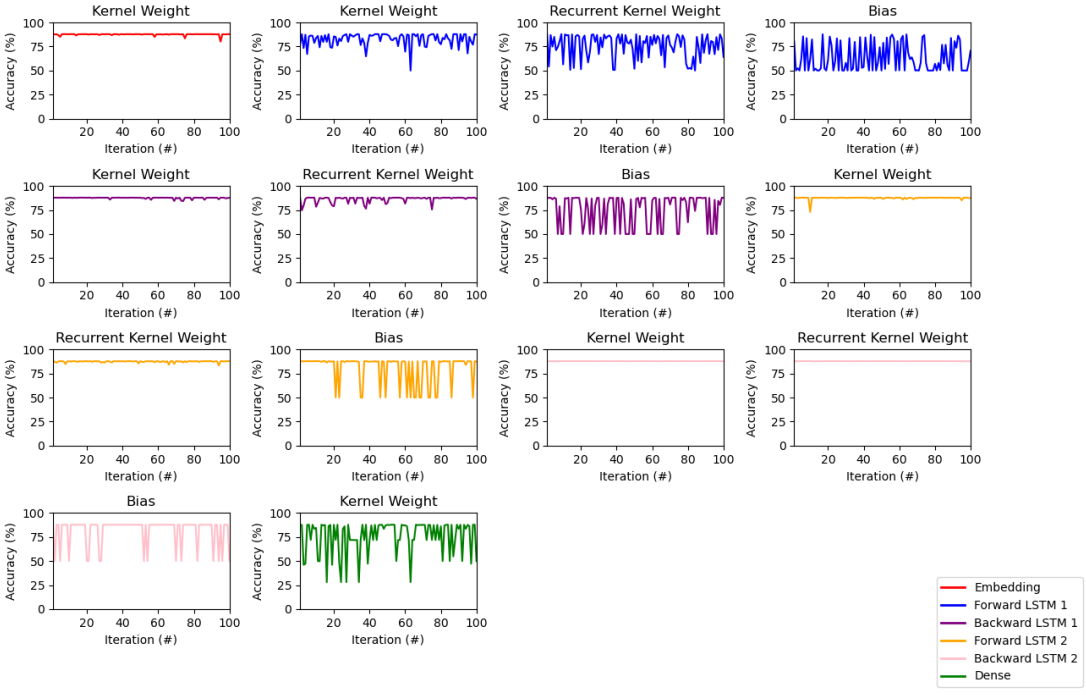


(b) Stacked LSTM Layers

Figure B.5. Original LSTM Models with 10 SEU in Exponent Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

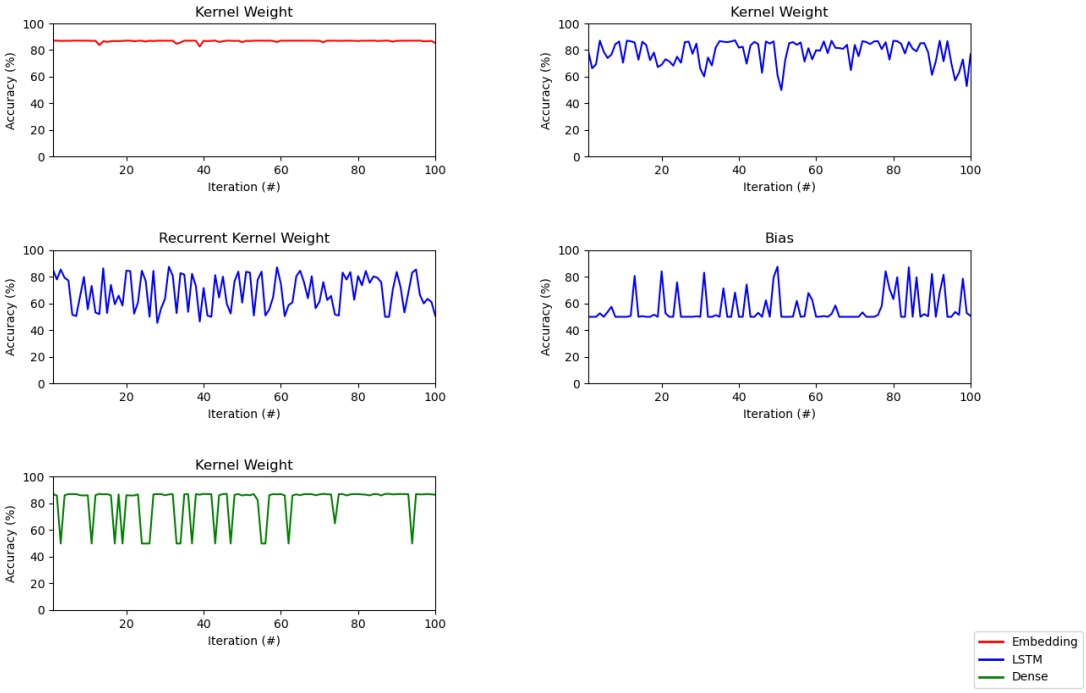


(c) Bidirectional LSTM Layer

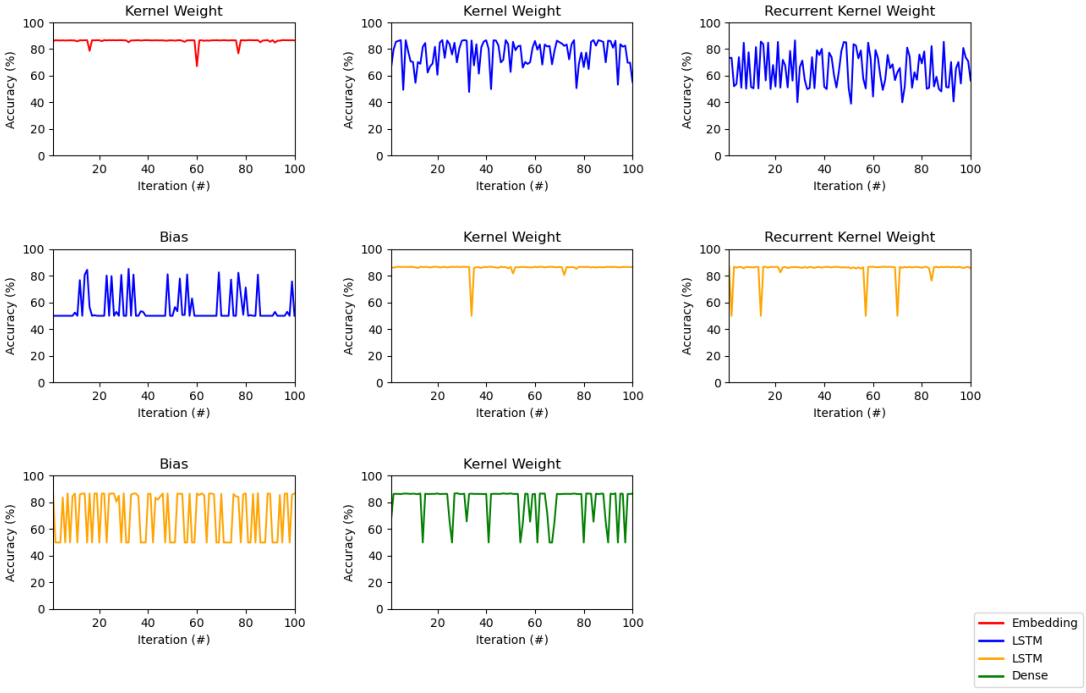


(d) Stacked Bidirectional LSTM Layers

Figure B.5. (cont'd) Original LSTM Models with 10 SEU in Exponent Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

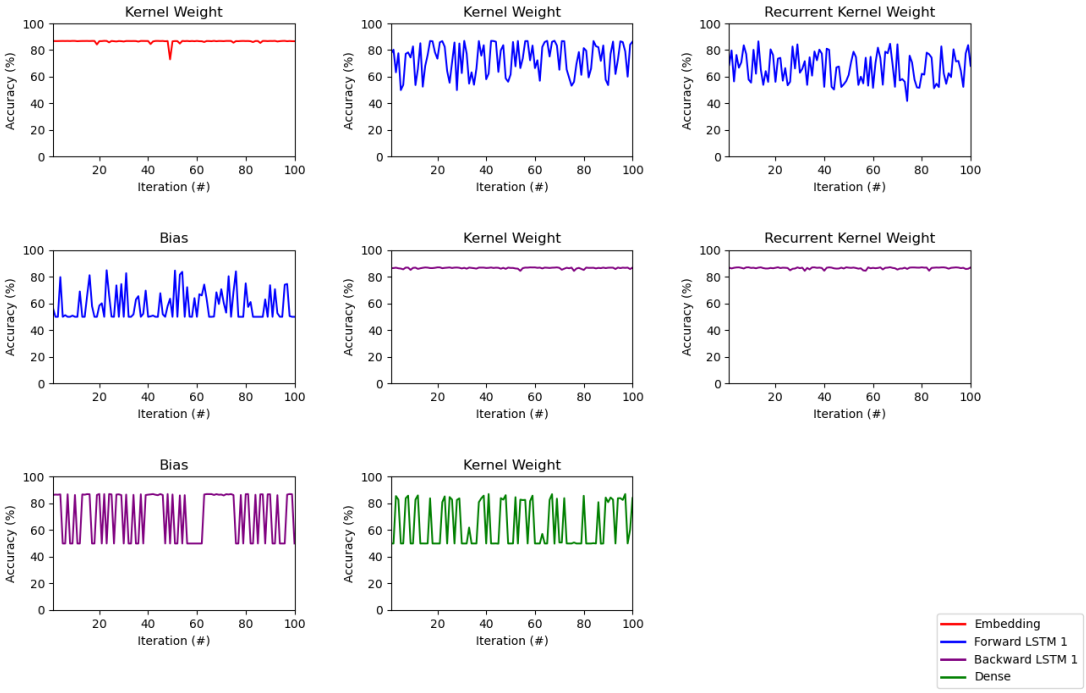


(a) LSTM Layer

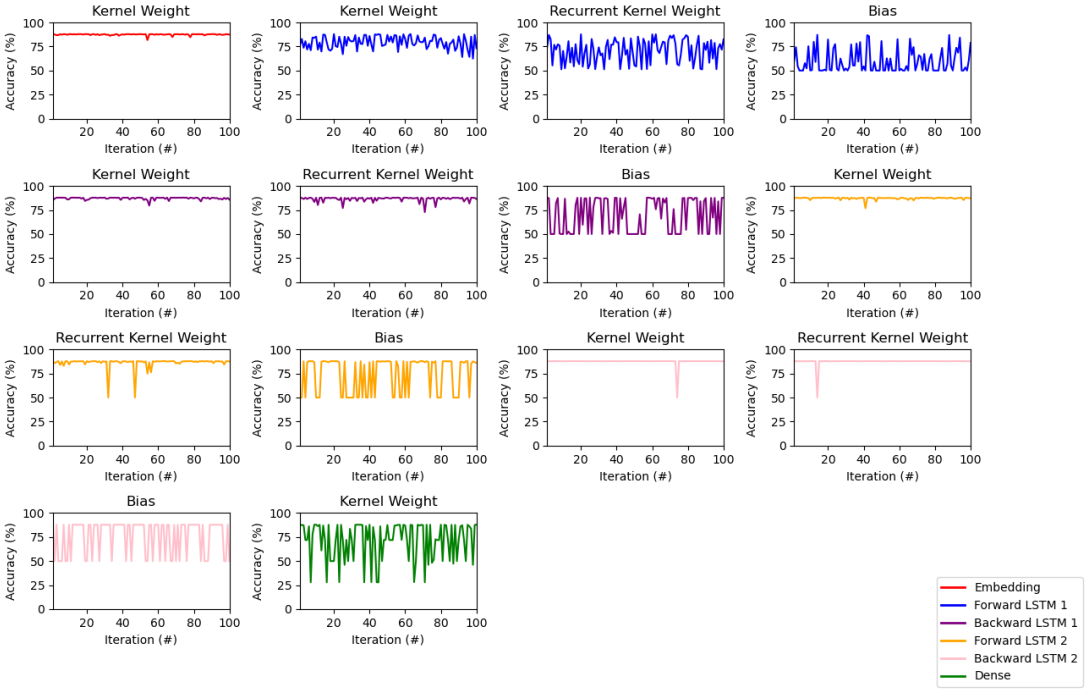


(b) Stacked LSTM Layers

Figure B.6. Original LSTM Models with 20 SEU in Exponent Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

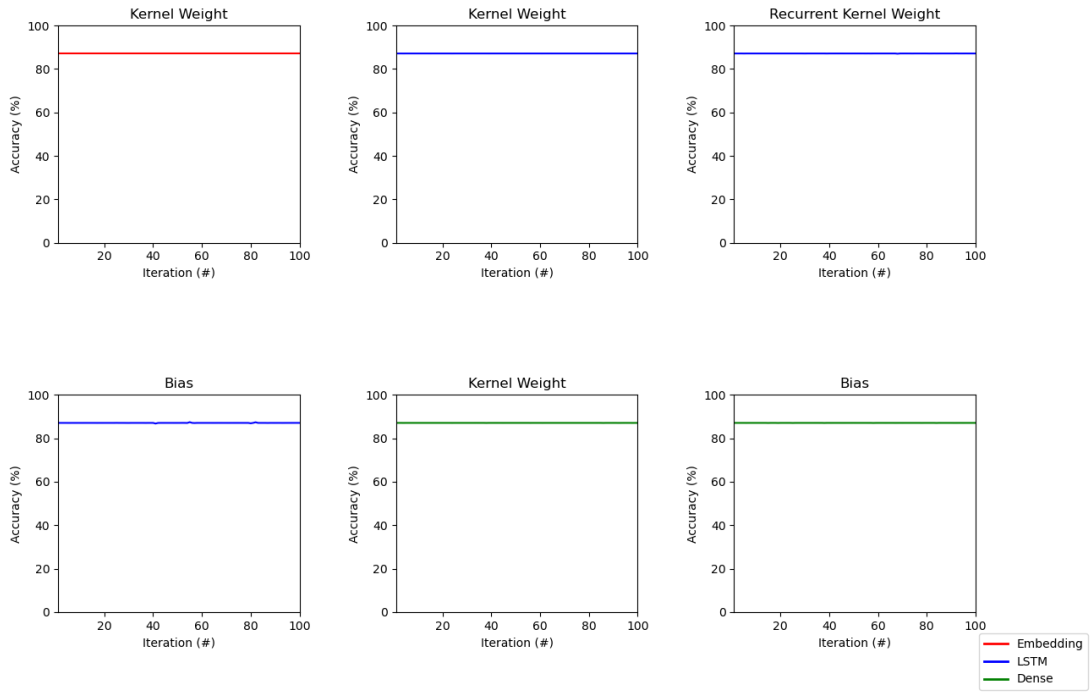


(c) Bidirectional LSTM Layer

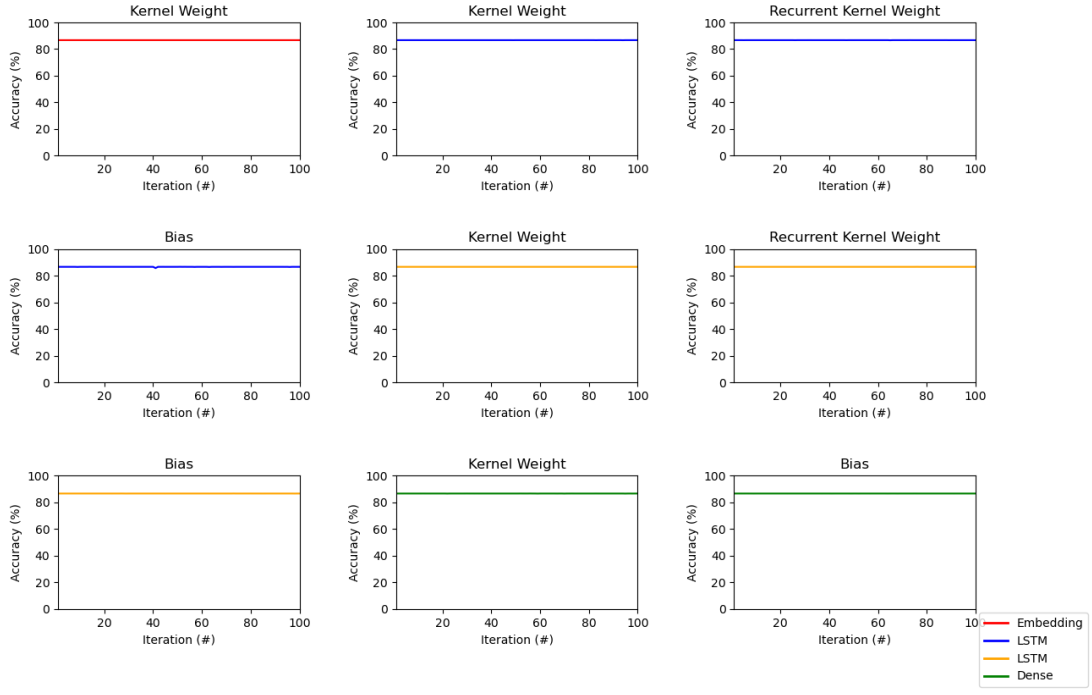


(d) Stacked Bidirectional LSTM Layers

Figure B.6. (cont'd) Original LSTM Models with 20 SEU in Exponent Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

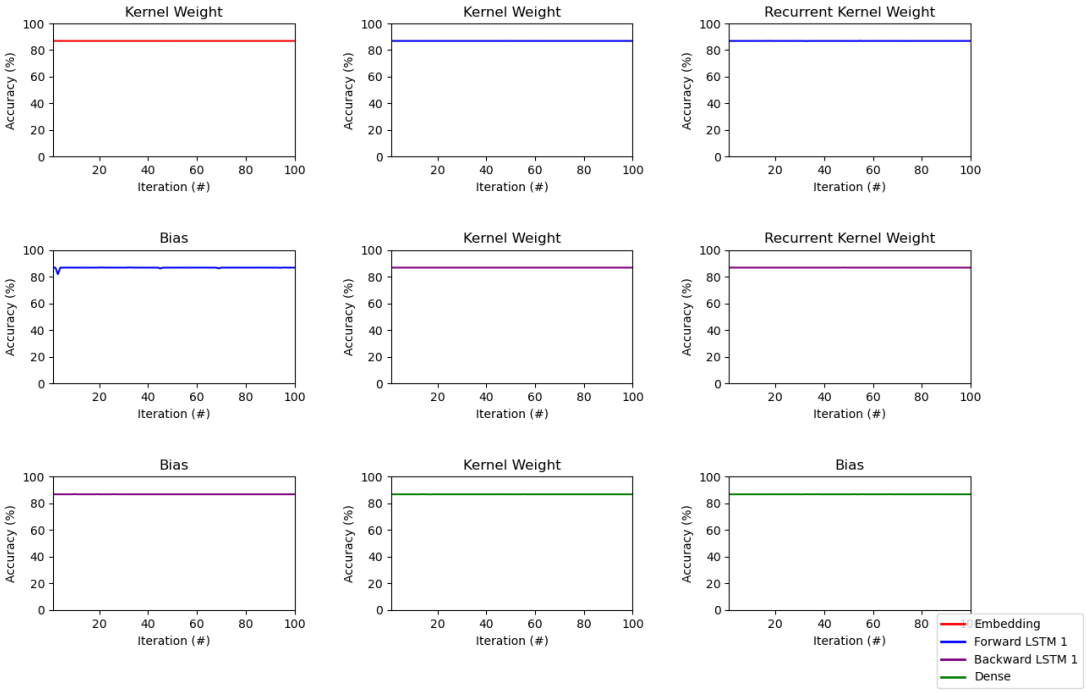


(a) LSTM Layer

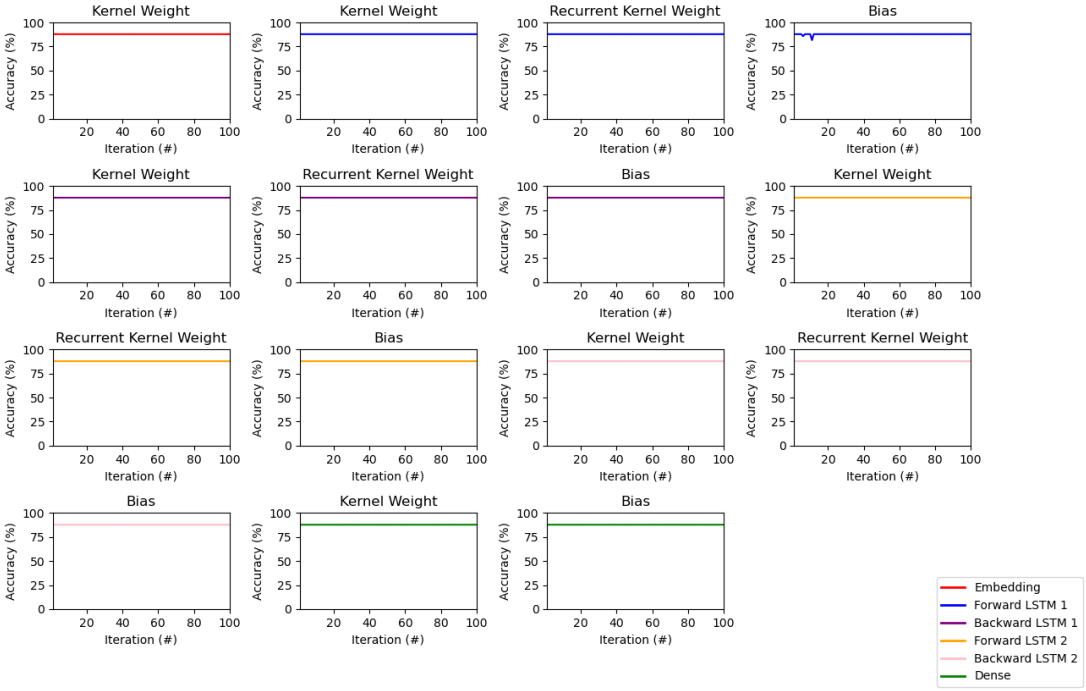


(b) Stacked LSTM Layers

Figure B.7. Original LSTM Models with 1 SEU in Mantissa Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

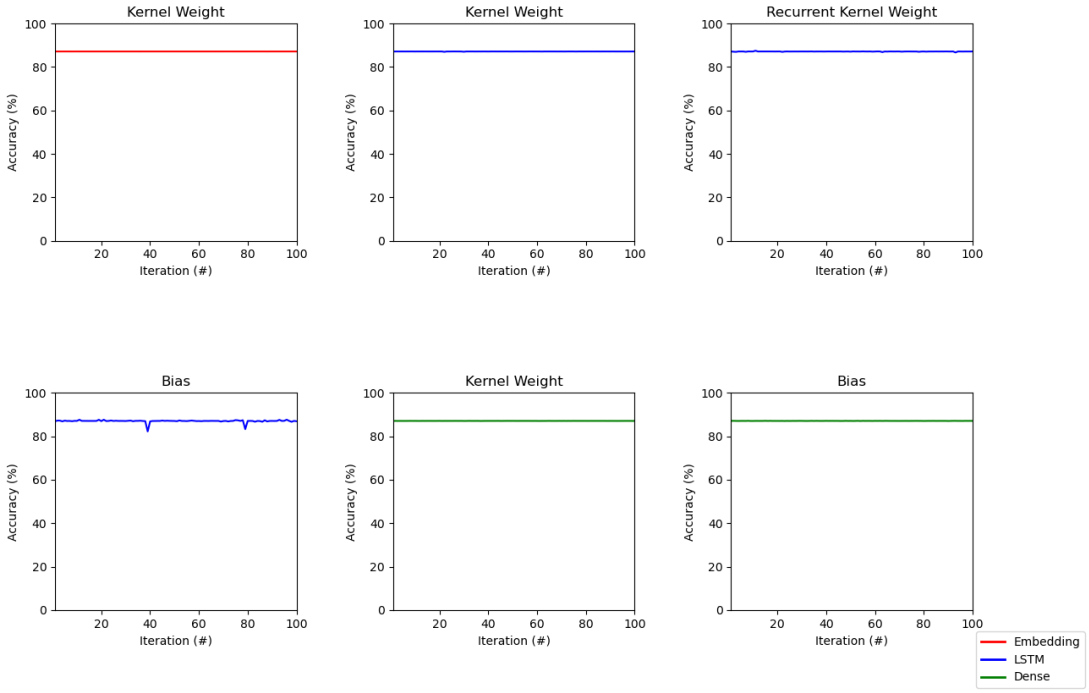


(c) Bidirectional LSTM Layer

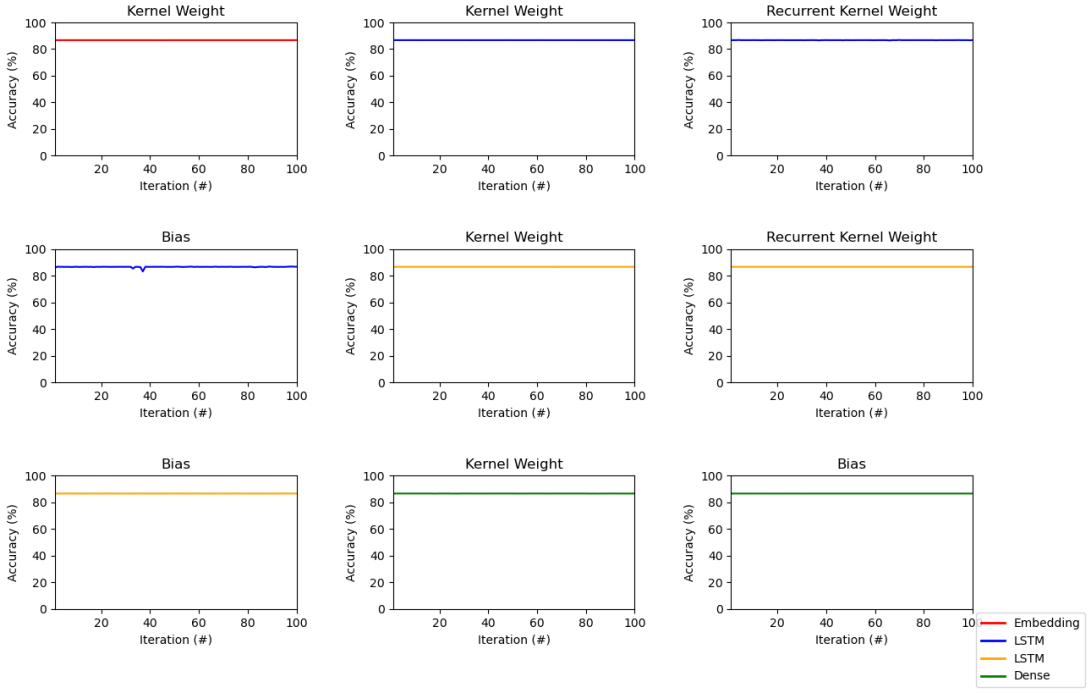


(d) Stacked Bidirectional LSTM Layers

Figure B.7. (cont'd) Original LSTM Models with 1 SEU in Mantissa Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

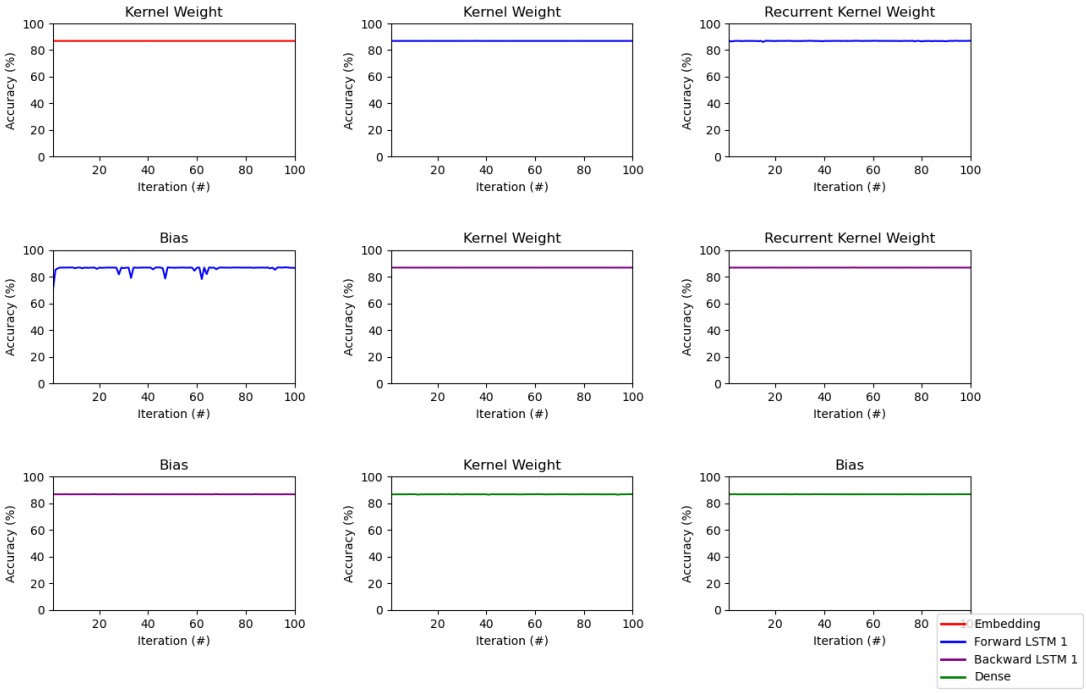


(a) LSTM Layer

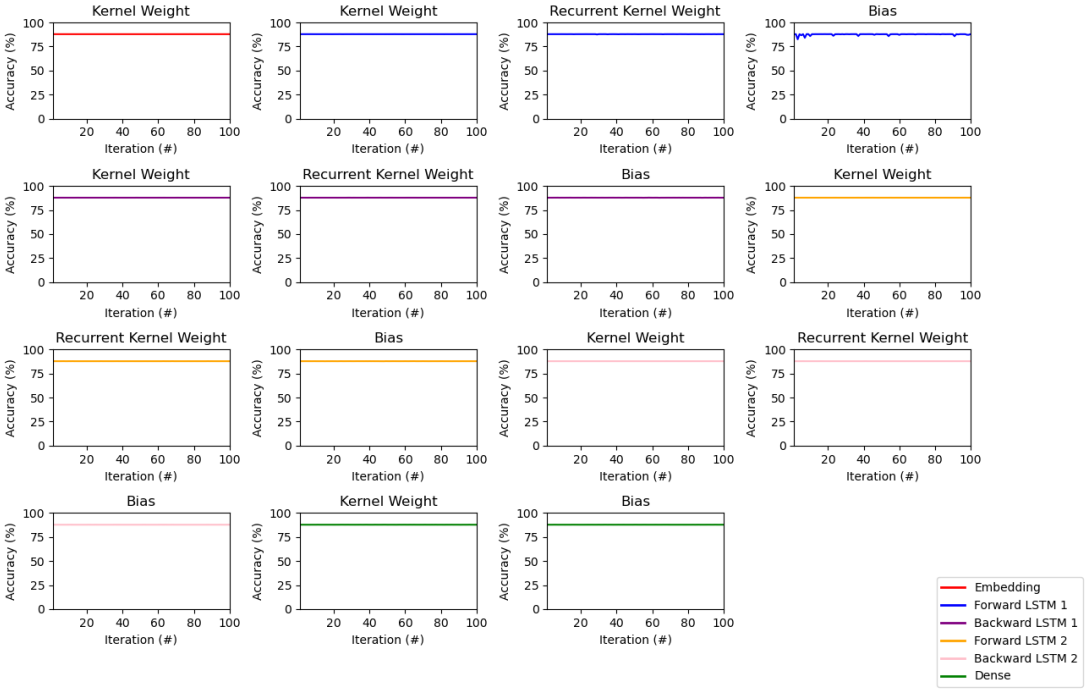


(b) Stacked LSTM Layers

Figure B.8. Original LSTM Models with 10 SEU in Mantissa Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

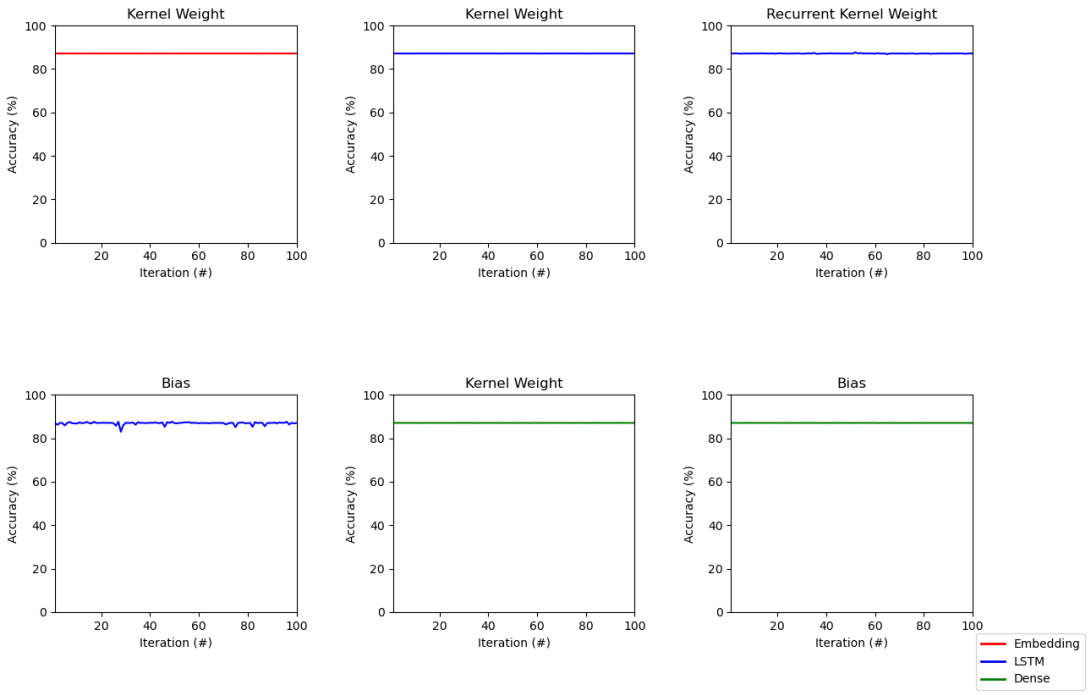


(c) Bidirectional LSTM Layer

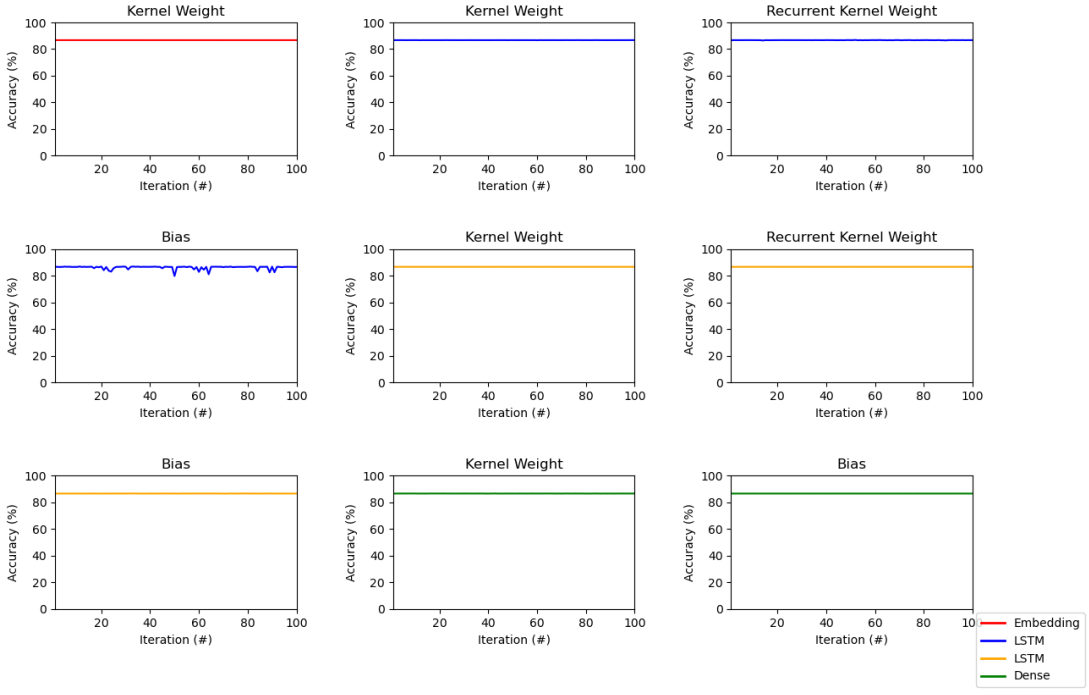


(d) Stacked Bidirectional LSTM Layers

Figure B.8. (cont'd) Original LSTM Models with 10 SEU in Mantissa Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

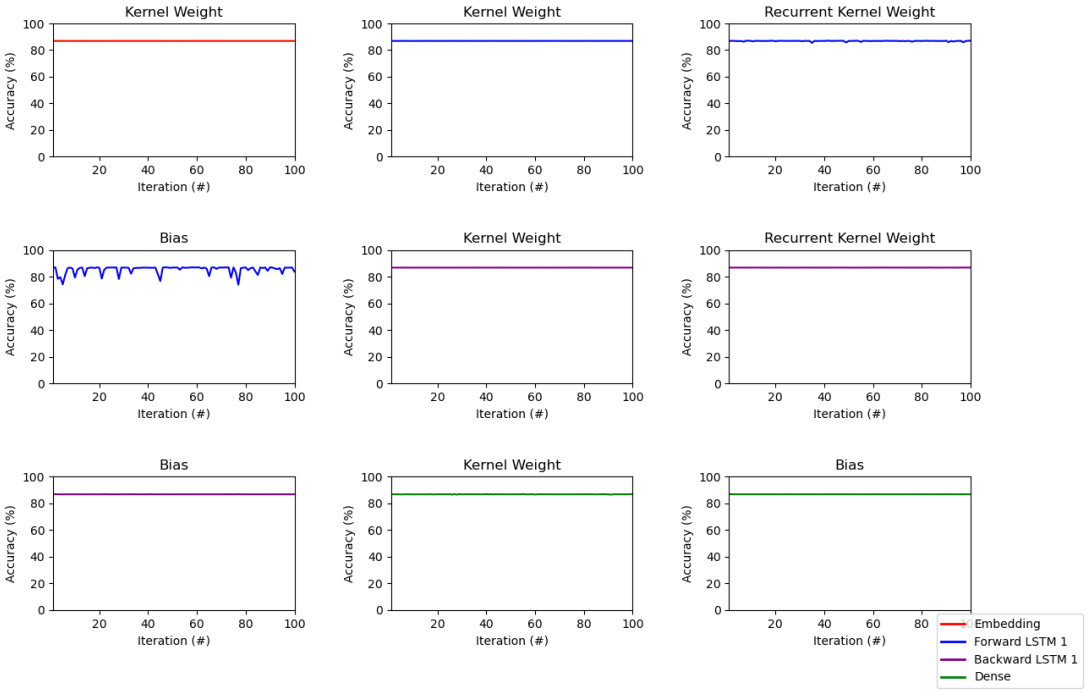


(a) LSTM Layer

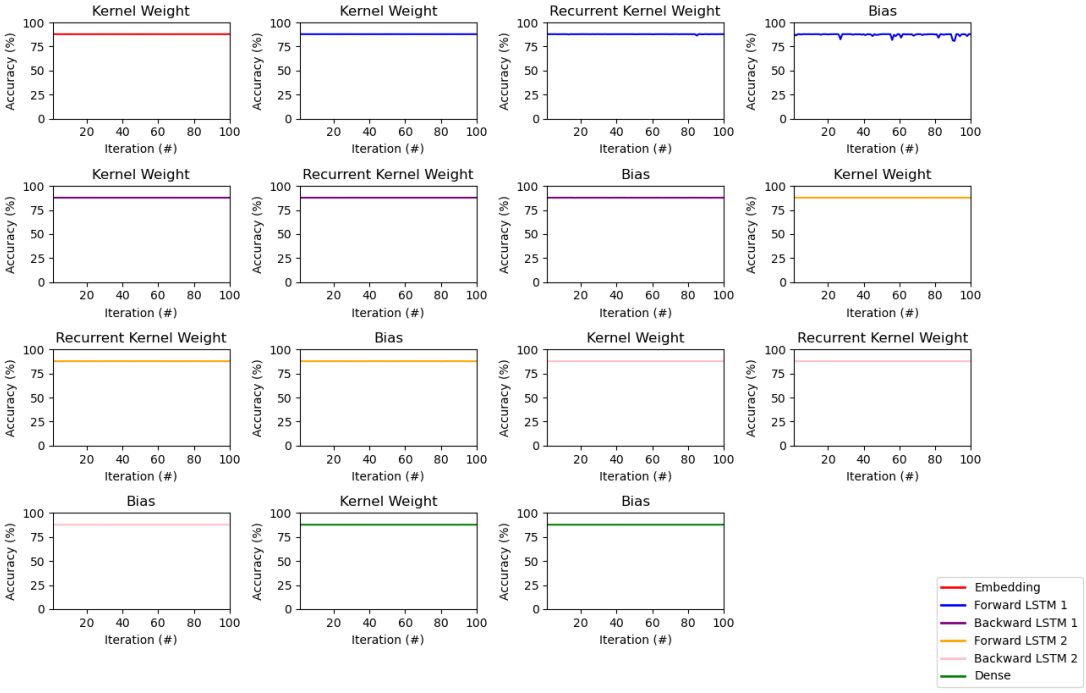


(b) Stacked LSTM Layers

Figure B.9. Original LSTM Models with 20 SEU in Mantissa Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

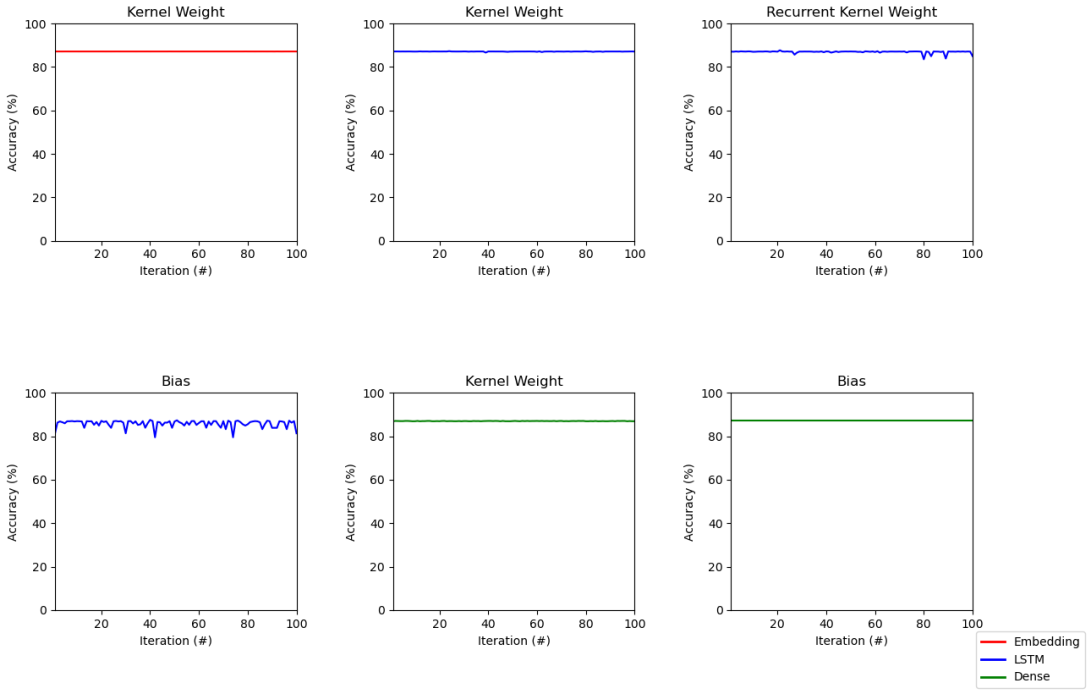


(c) Bidirectional LSTM Layer

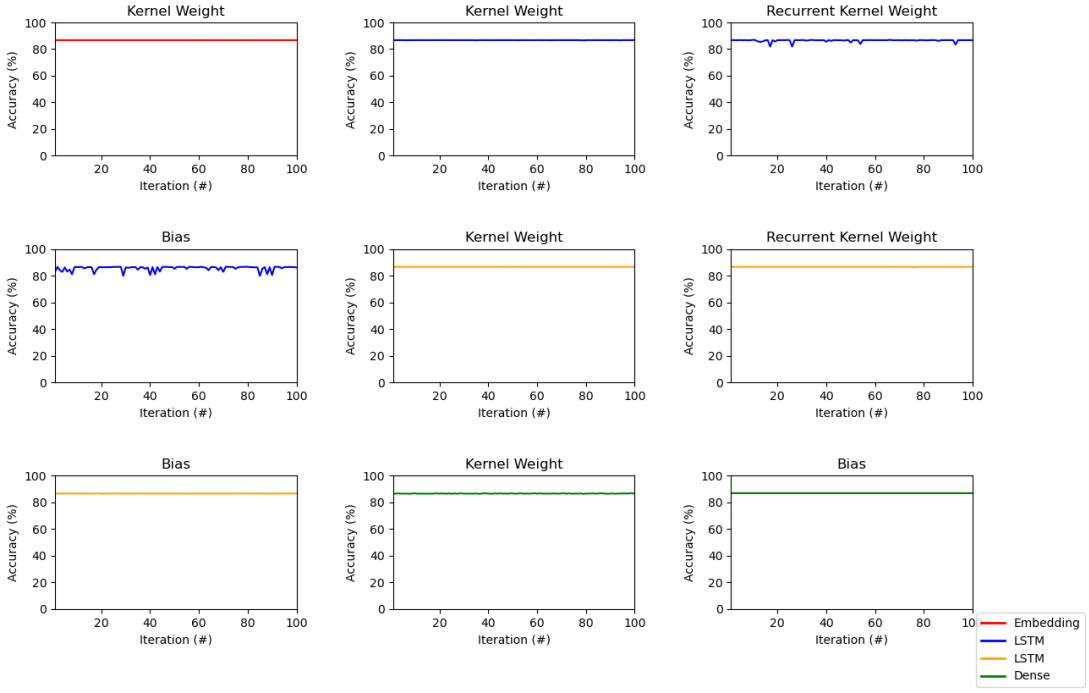


(d) Stacked Bidirectional LSTM Layers

Figure B.9. (cont'd) Original LSTM Models with 20 SEU in Mantissa Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

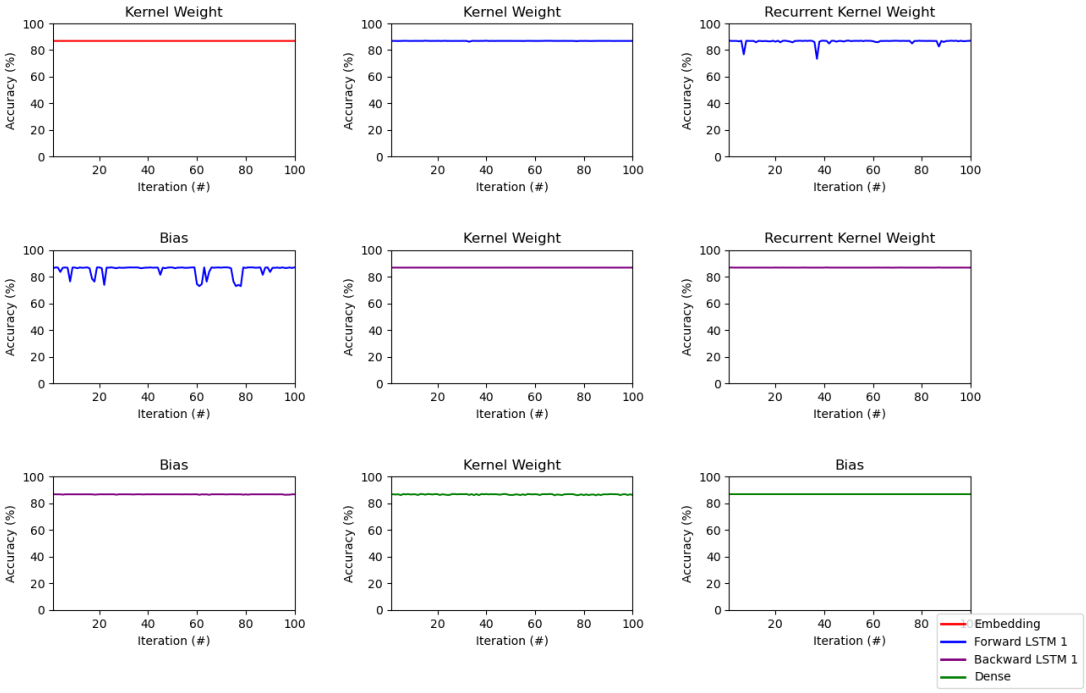


(a) LSTM Layer

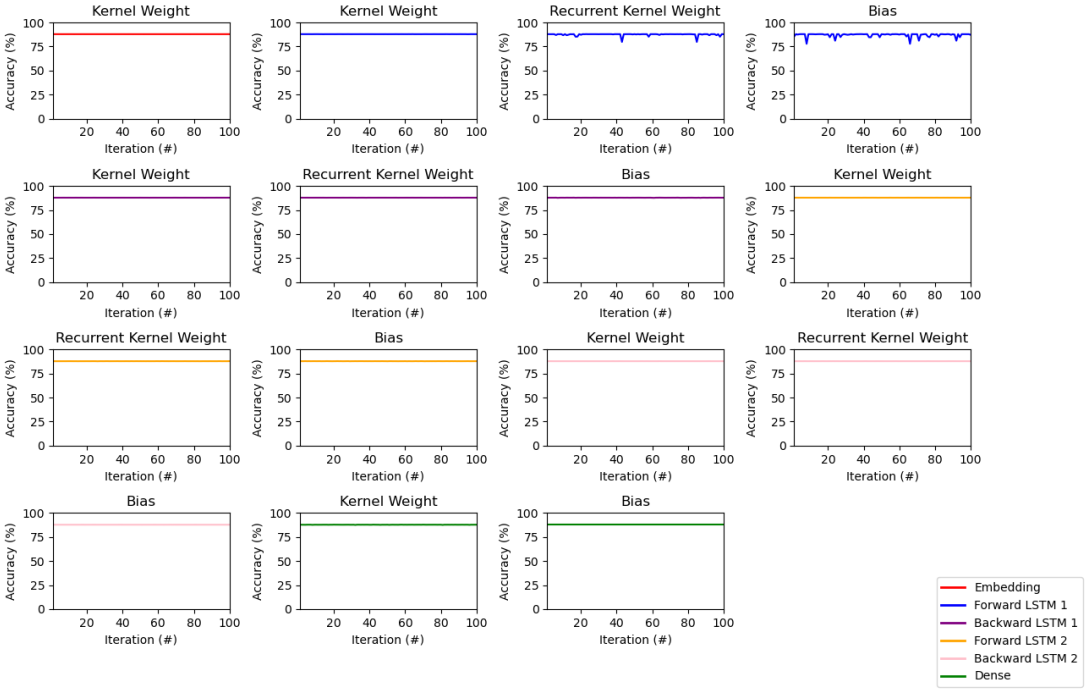


(b) Stacked LSTM Layers

Figure B.10. Original LSTM Models with 1 SEU in Sign Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

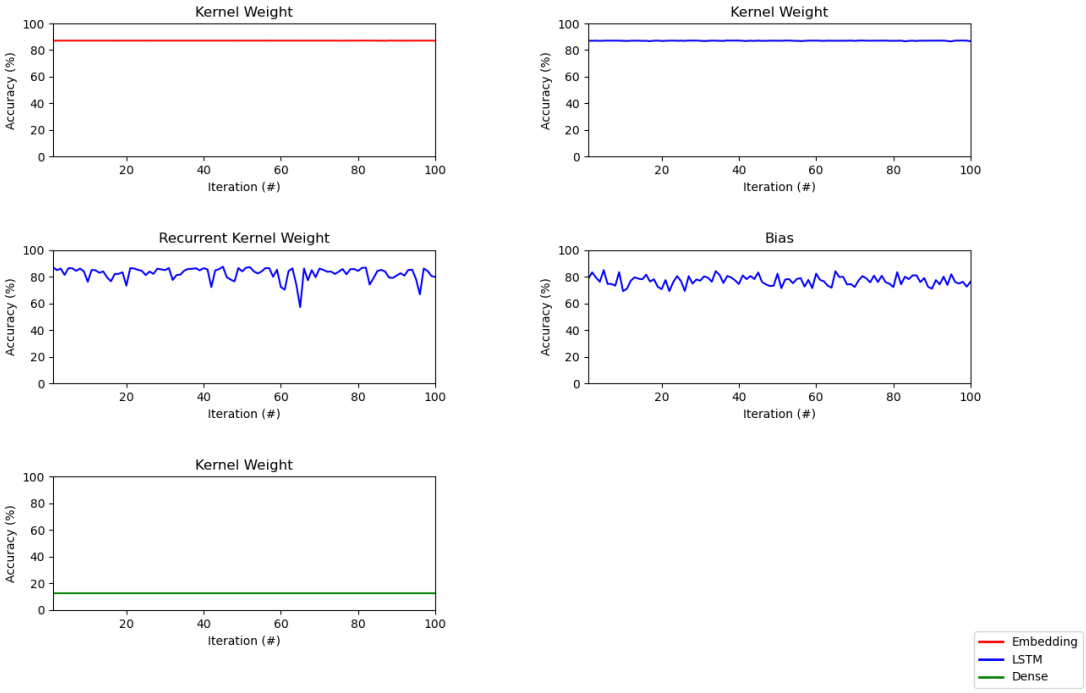


(c) Bidirectional LSTM Layer

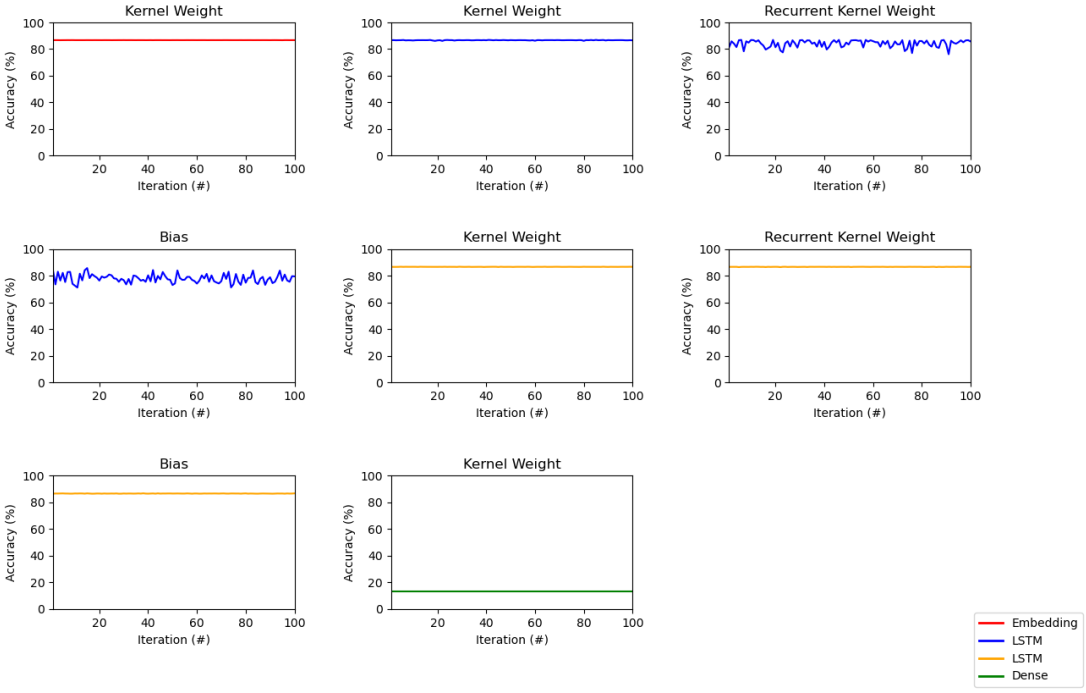


(d) Stacked Bidirectional LSTM Layers

Figure B.10. (cont'd) Original LSTM Models with 1 SEU in Sign Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

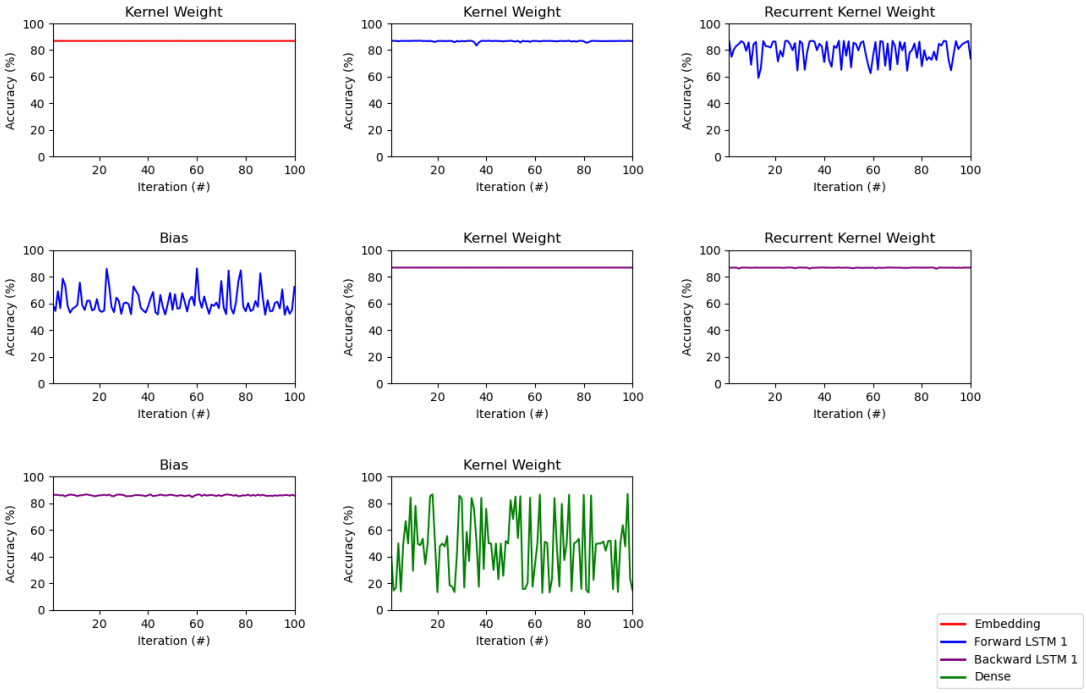


(a) LSTM Layer

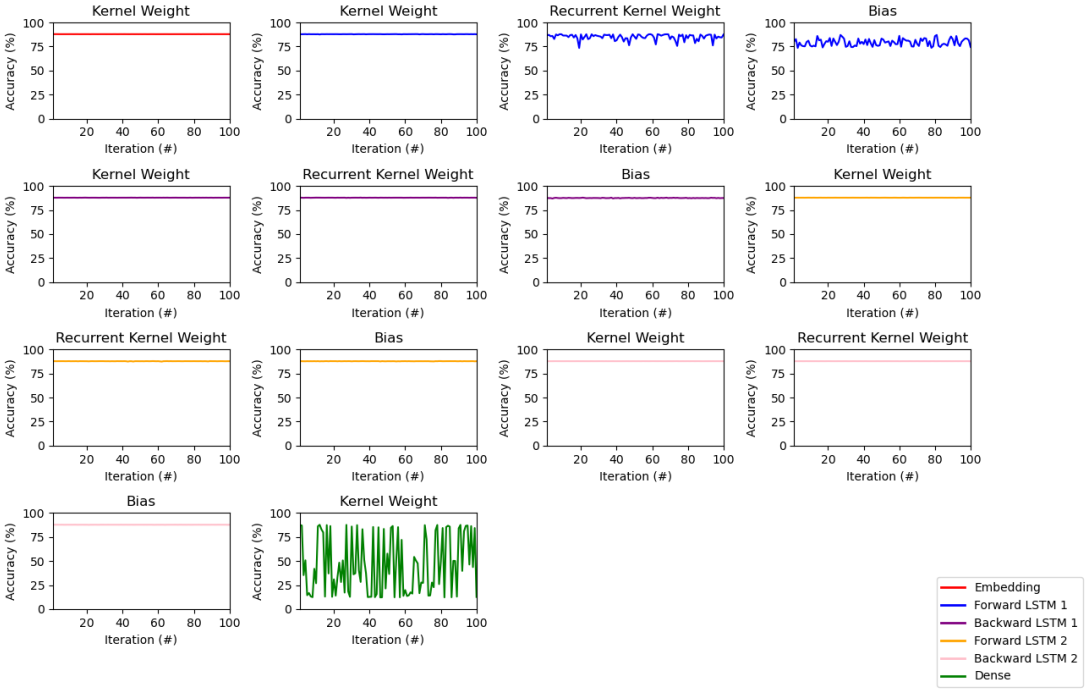


(b) Stacked LSTM Layers

Figure B.11. Original LSTM Models with 10 SEU in Sign Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

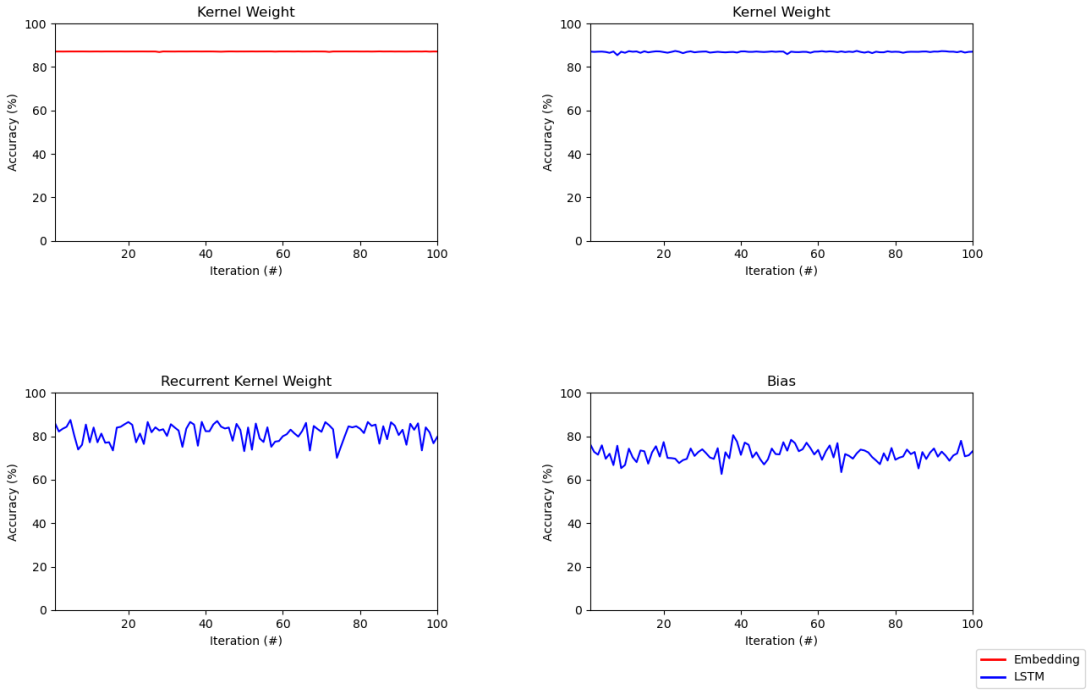


(c) Bidirectional LSTM Layer

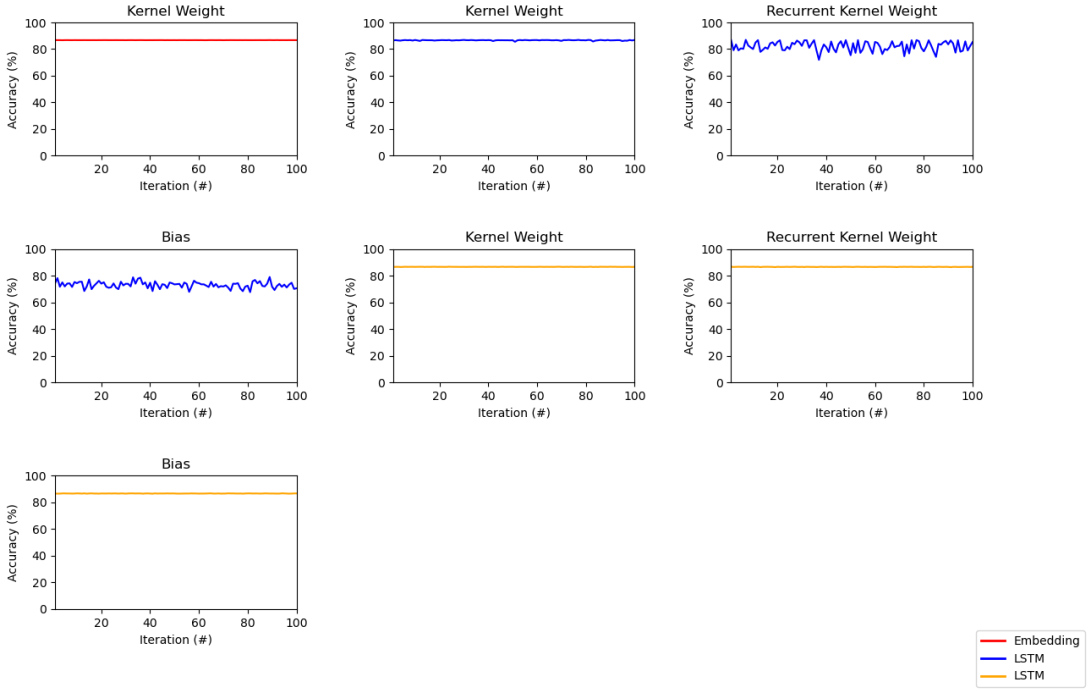


(d) Stacked Bidirectional LSTM Layers

Figure B.11. (cont'd) Original LSTM Models with 10 SEU in Sign Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

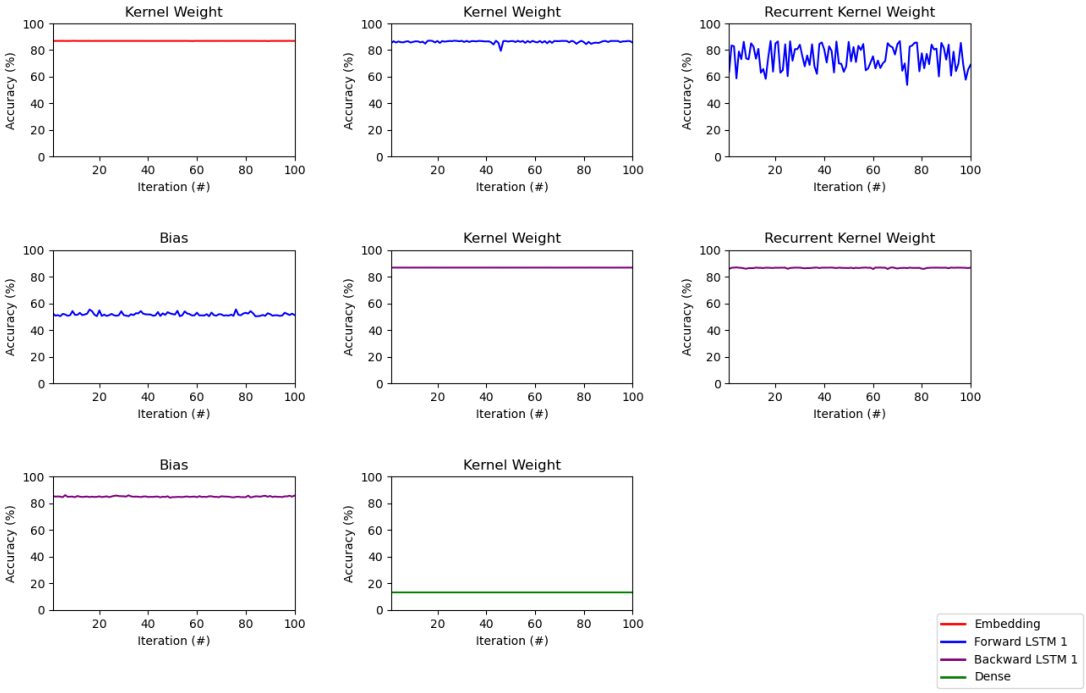


(a) LSTM Layer

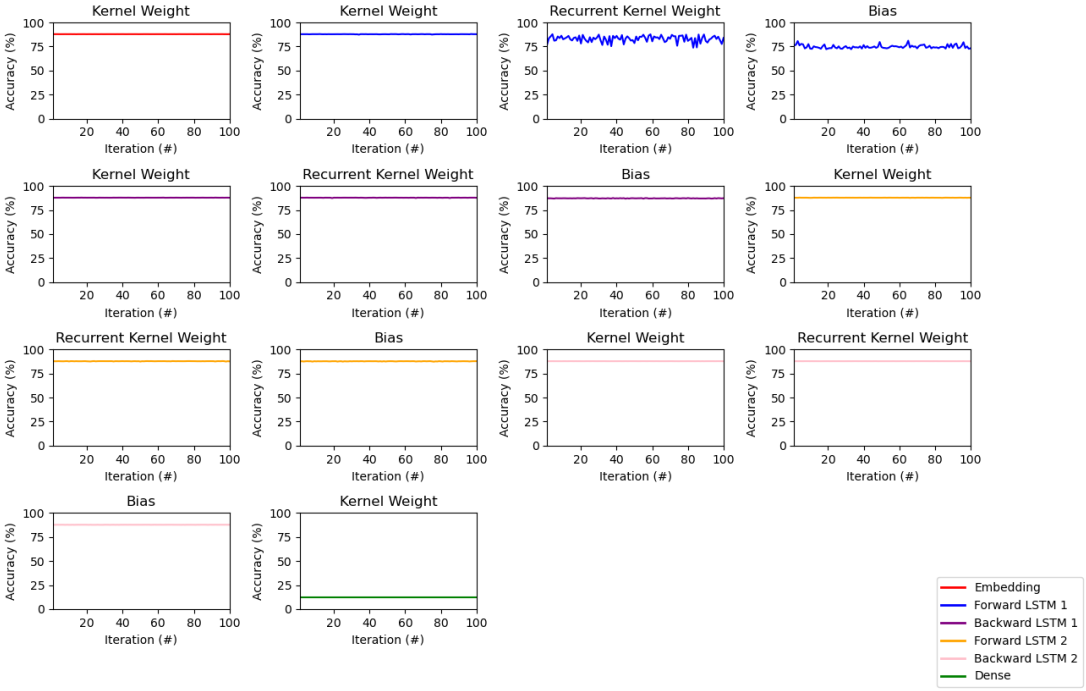


(b) Stacked LSTM Layers

Figure B.12. Original LSTM Models with 20 SEU in Sign Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations



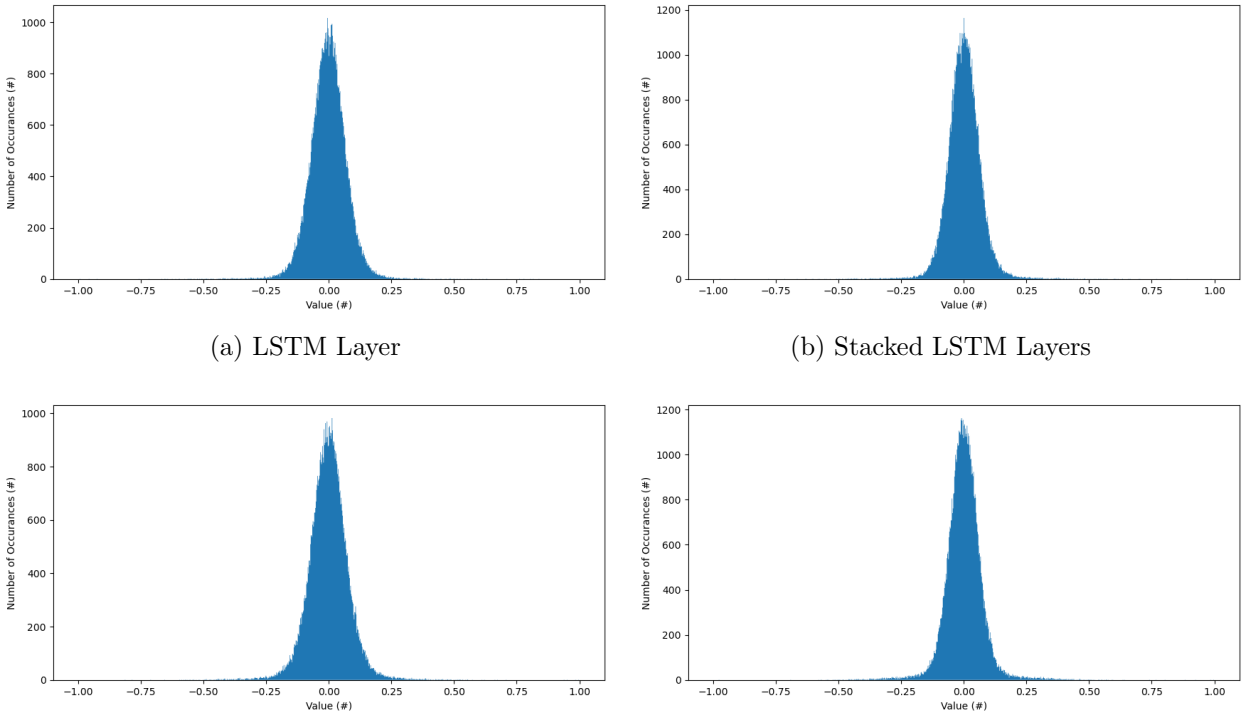
(c) Bidirectional LSTM Layer



(d) Stacked Bidirectional LSTM Layers

Figure B.12. (cont'd) Original LSTM Models with 20 SEU in Sign Bit Field of Each Set of Weights/Biases in Each Layer Over 100 Iterations

Appendix C. Supplemental Figure on Mitigated & Modified LSTM Networks



(a) LSTM Layer

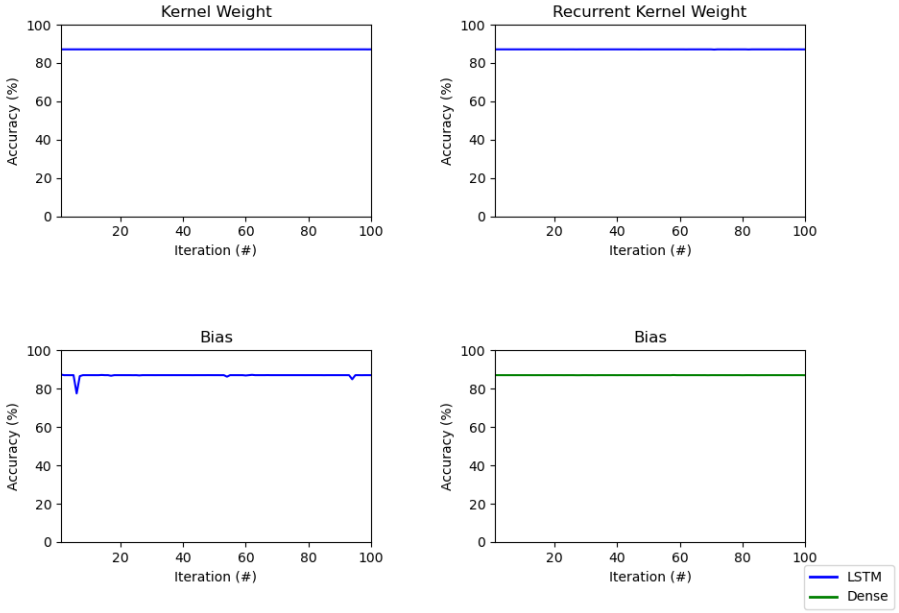
(b) Stacked LSTM Layers

(c) Bidirectional LSTM Layer

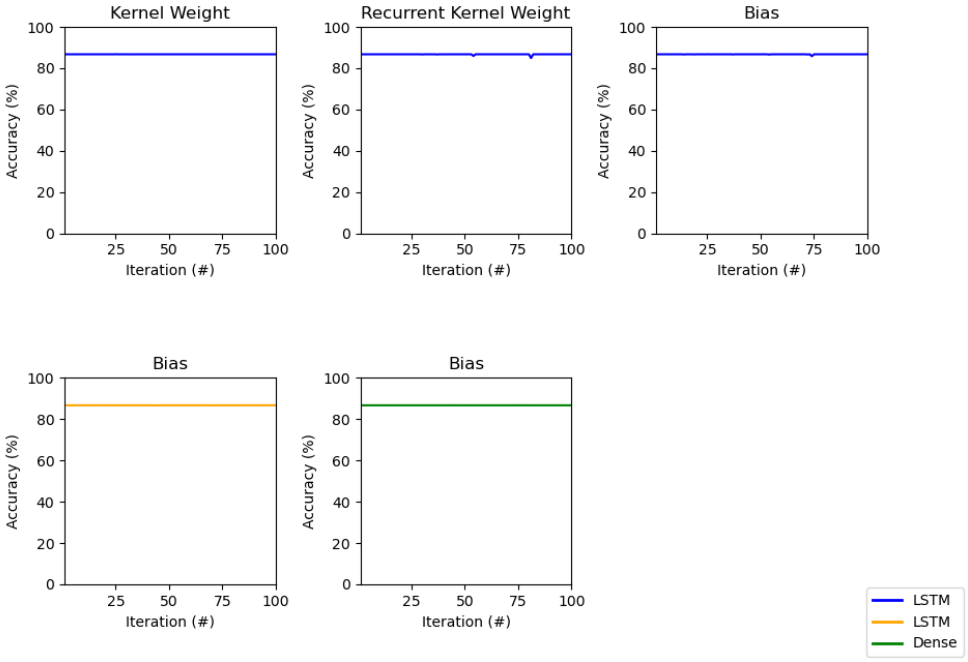
(d) Stacked Bidirectional LSTM Layers

Figure C.1. Distribution of Trained Weights & Biases of the Modified LSTM Networks

Appendix D. Supplemental Results of Experiment 2 Injection Results

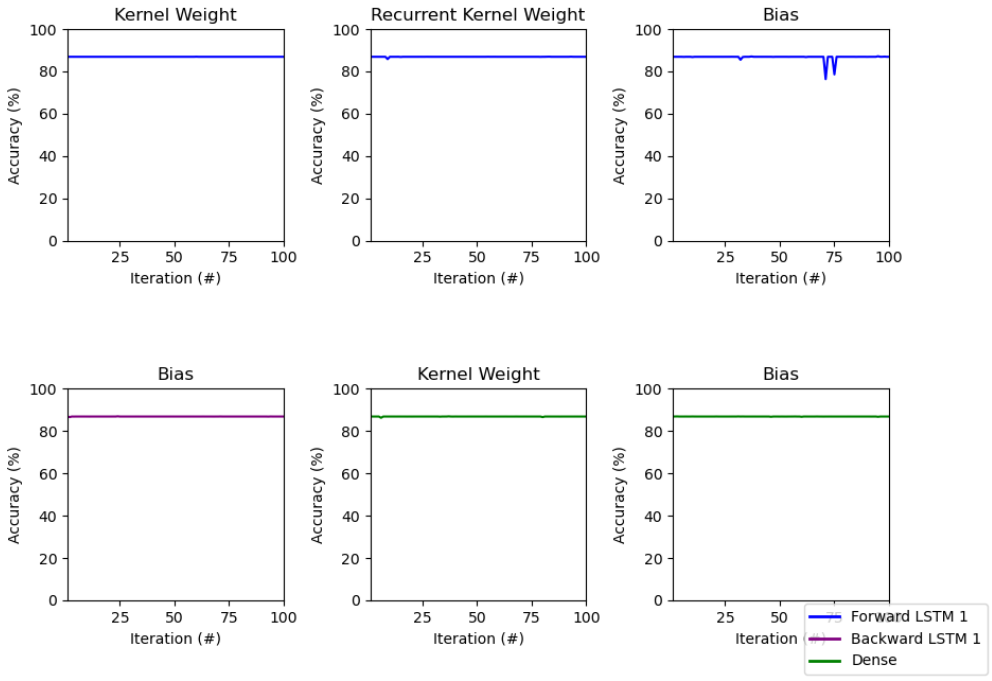


(a) LSTM Layer

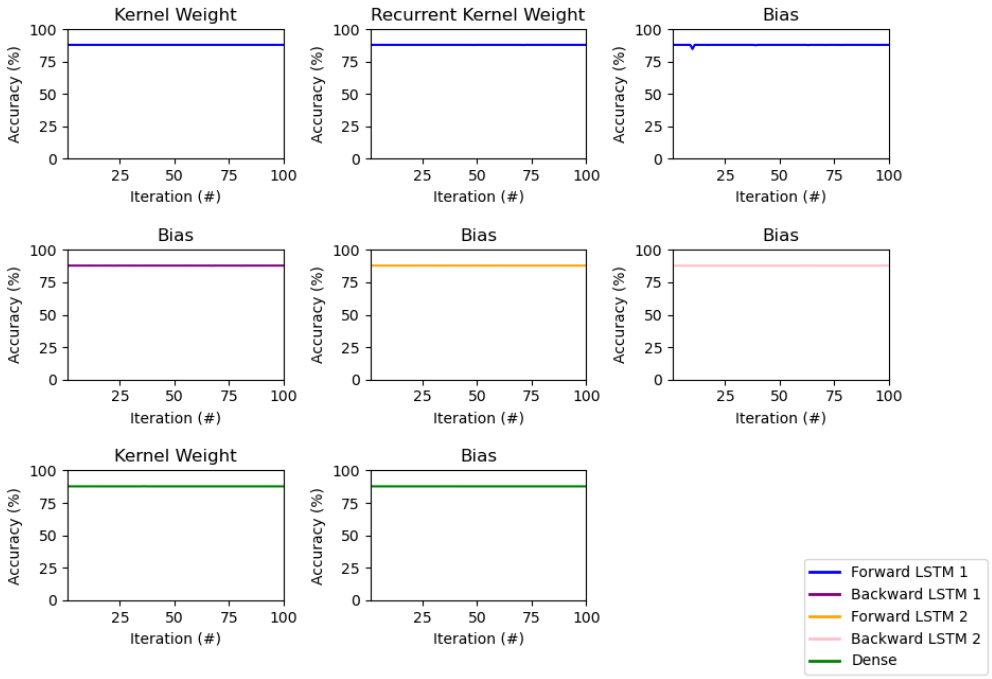


(b) Stacked LSTM Layers

Figure D.1. Modified LSTM Models with 1 SEU in Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

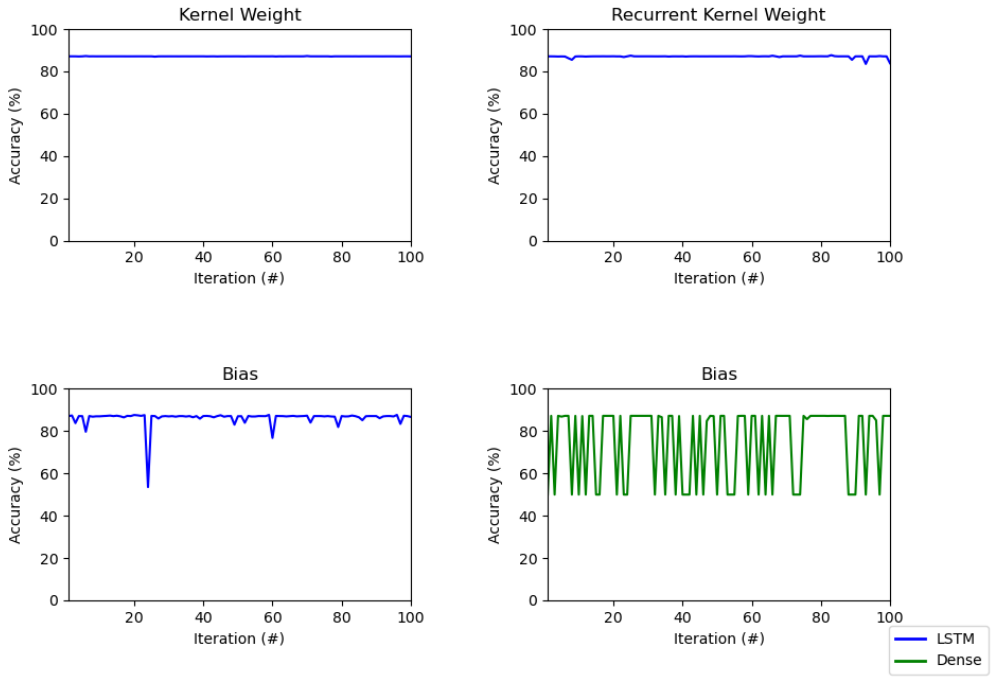


(c) Bidirectional LSTM Layer

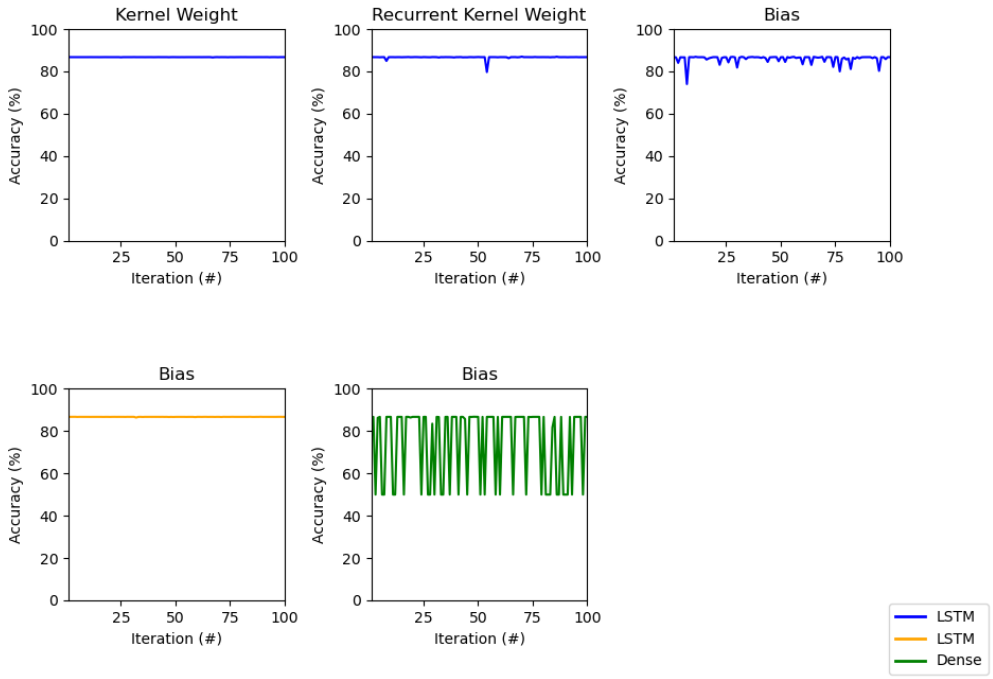


(d) Stacked Bidirectional LSTM Layers

Figure D.1. (cont'd) Modified LSTM Models with 1 SEU in Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

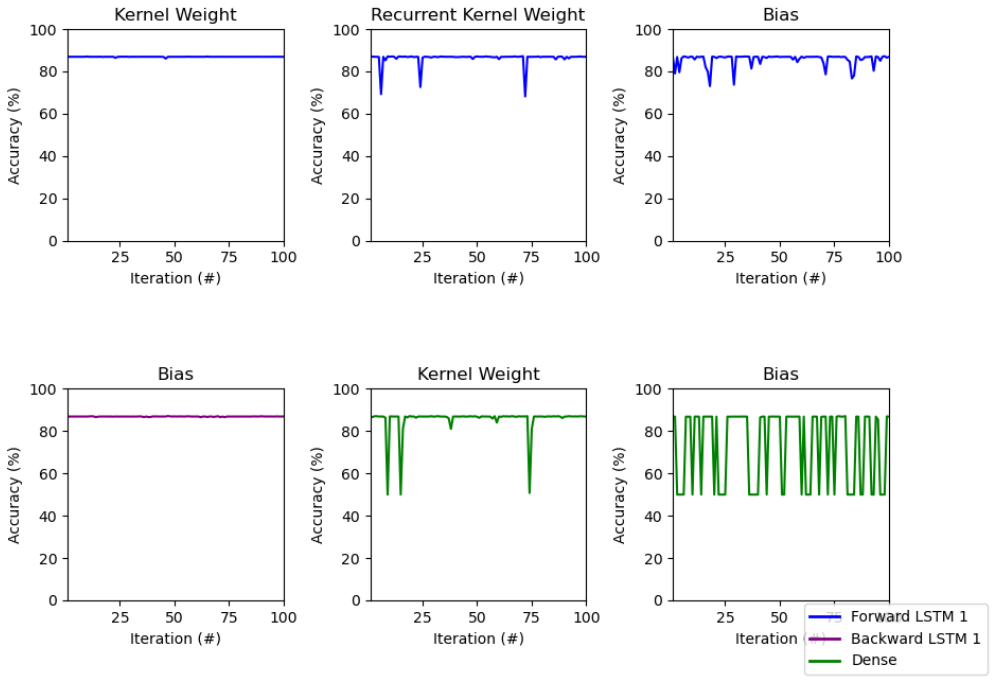


(a) LSTM Layer

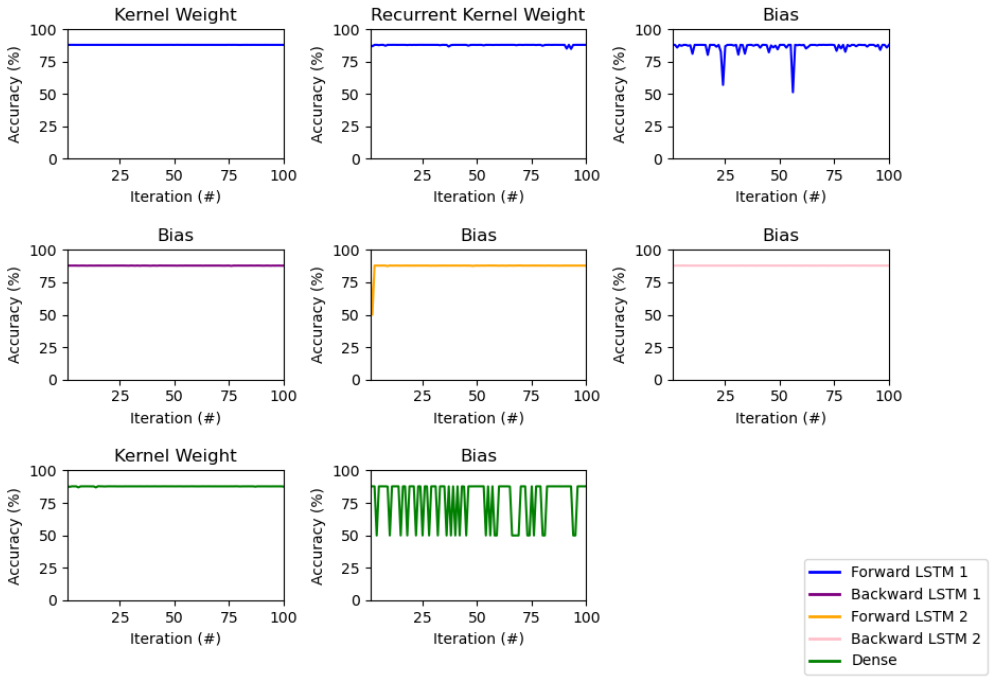


(b) Stacked LSTM Layers

Figure D.2. Modified LSTM Models with 10 SEU in Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

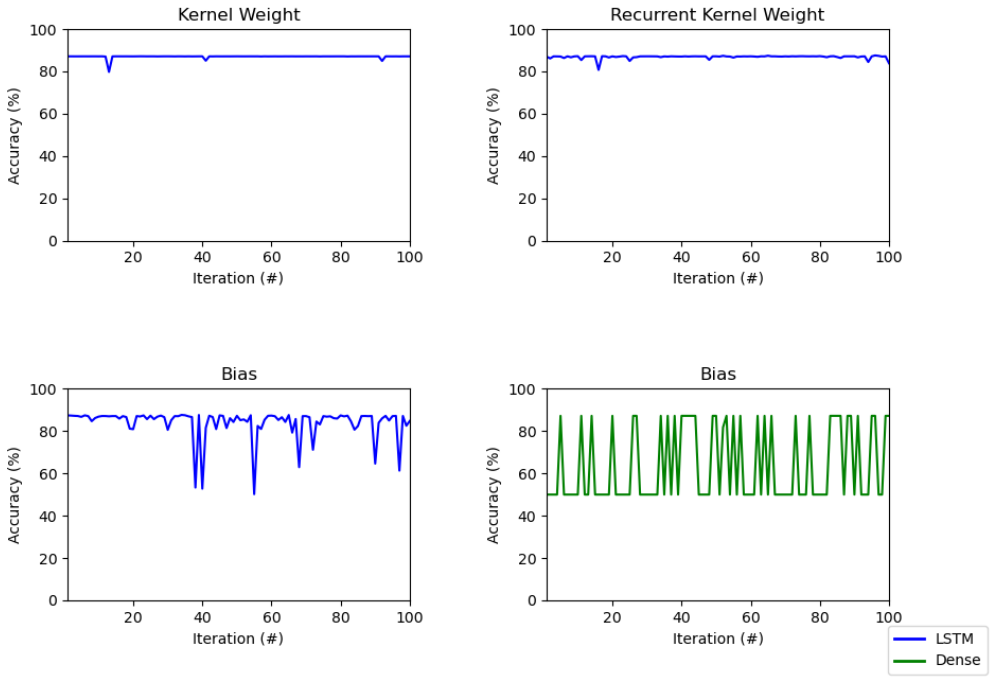


(c) Bidirectional LSTM Layer

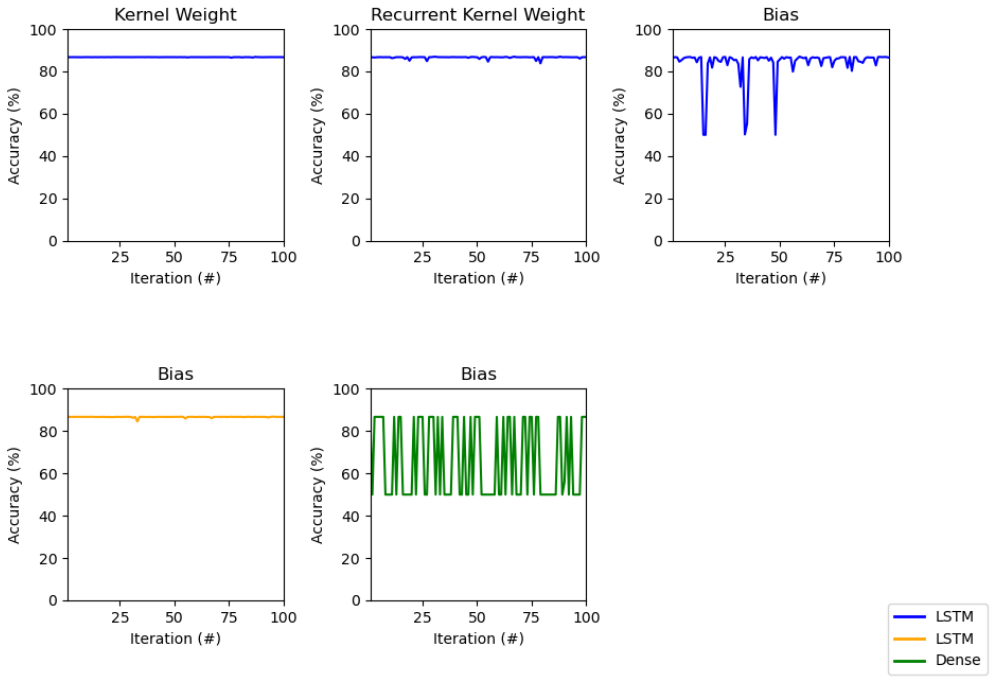


(d) Stacked Bidirectional LSTM Layers

Figure D.2. (cont'd) Modified LSTM Models with 10 SEU in Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

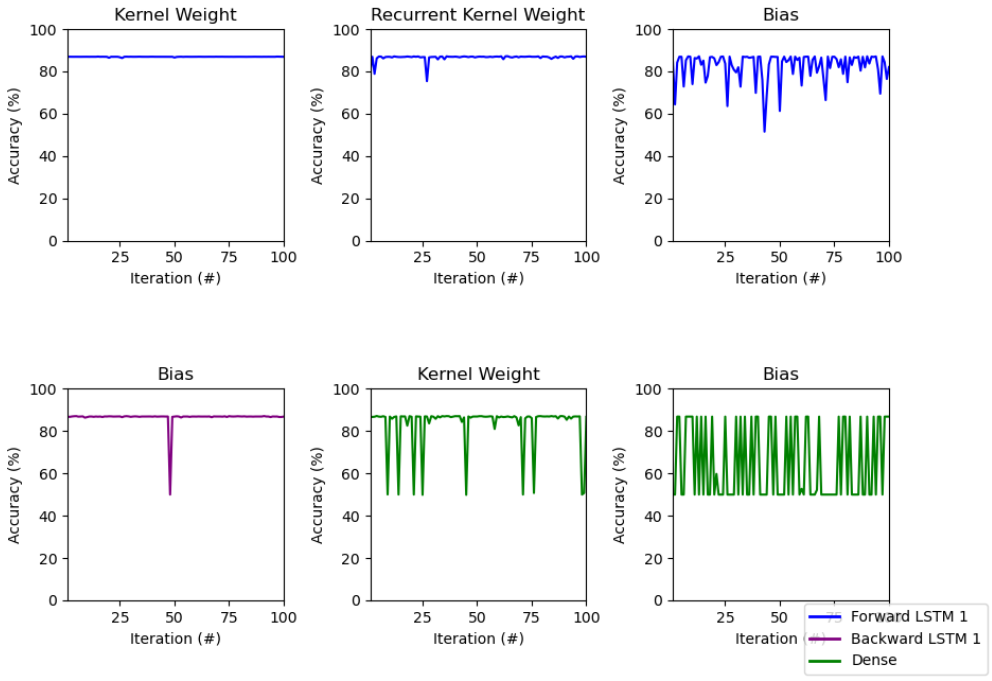


(a) LSTM Layer

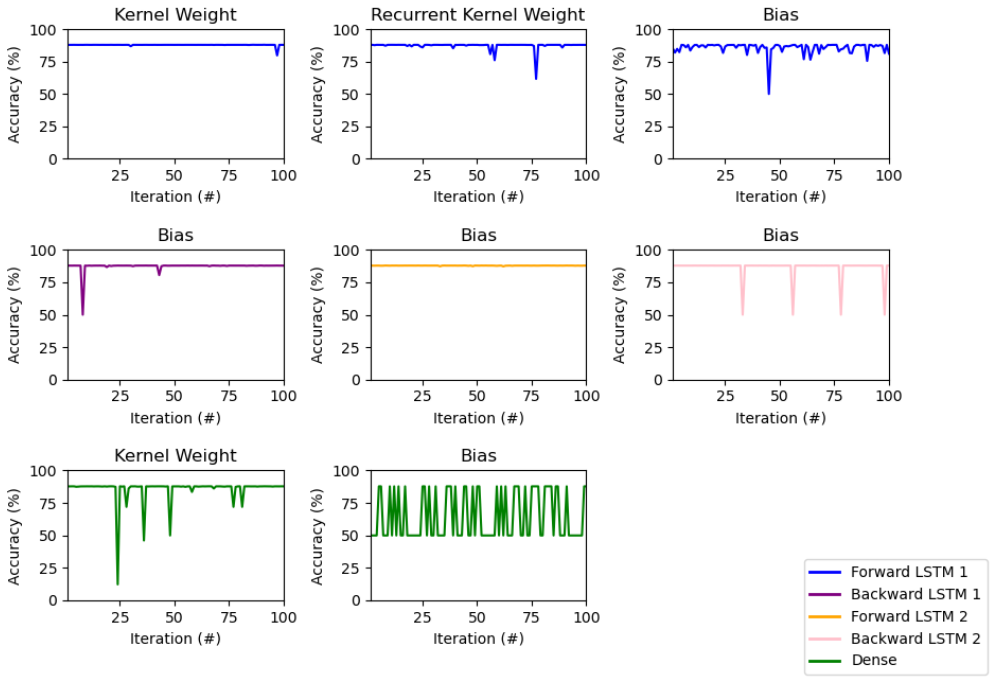


(b) Stacked LSTM Layers

Figure D.3. Modified LSTM Models with 20 SEU in Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

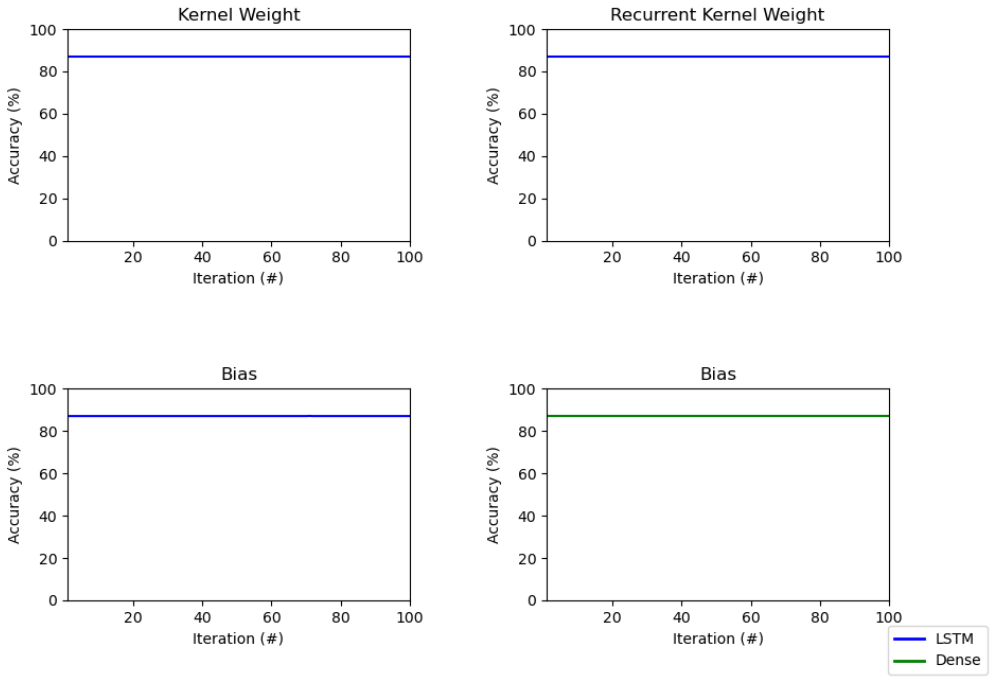


(c) Bidirectional LSTM Layer

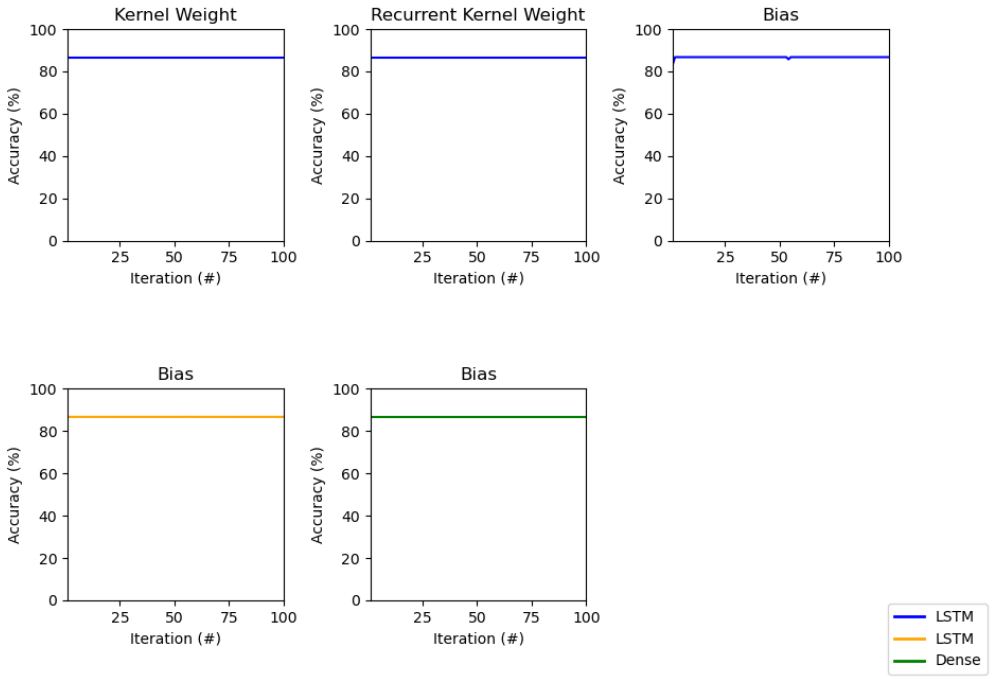


(d) Stacked Bidirectional LSTM Layers

Figure D.3. (cont'd) Modified LSTM Models with 20 SEU in Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

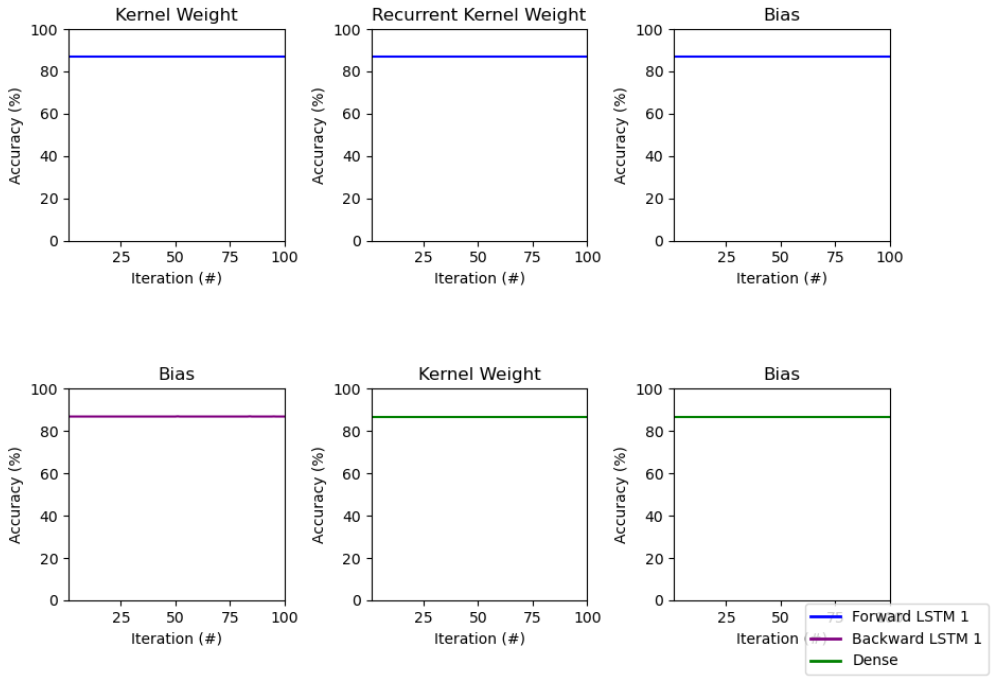


(a) LSTM Layer

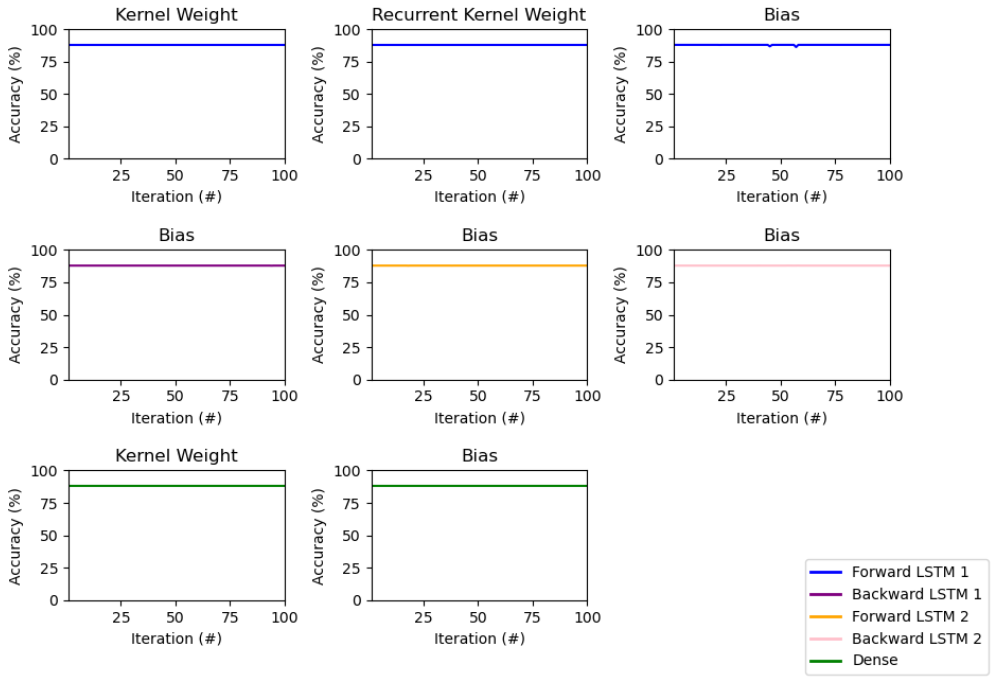


(b) Stacked LSTM Layers

Figure D.4. Modified LSTM Models with 1 SEU in Exponent Bit Field of Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

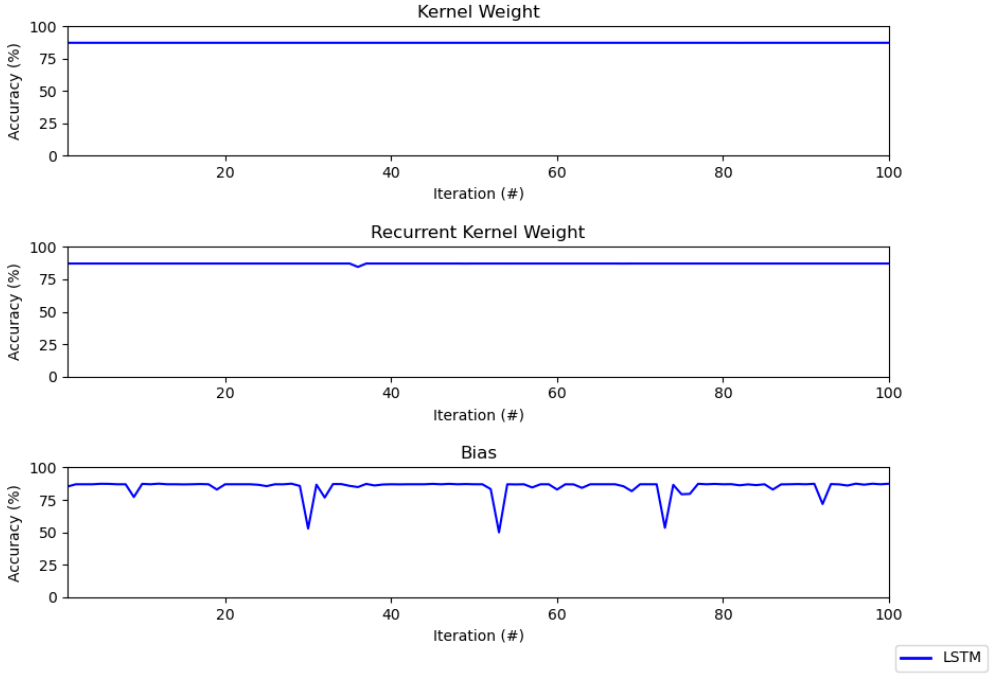


(c) Bidirectional LSTM Layer

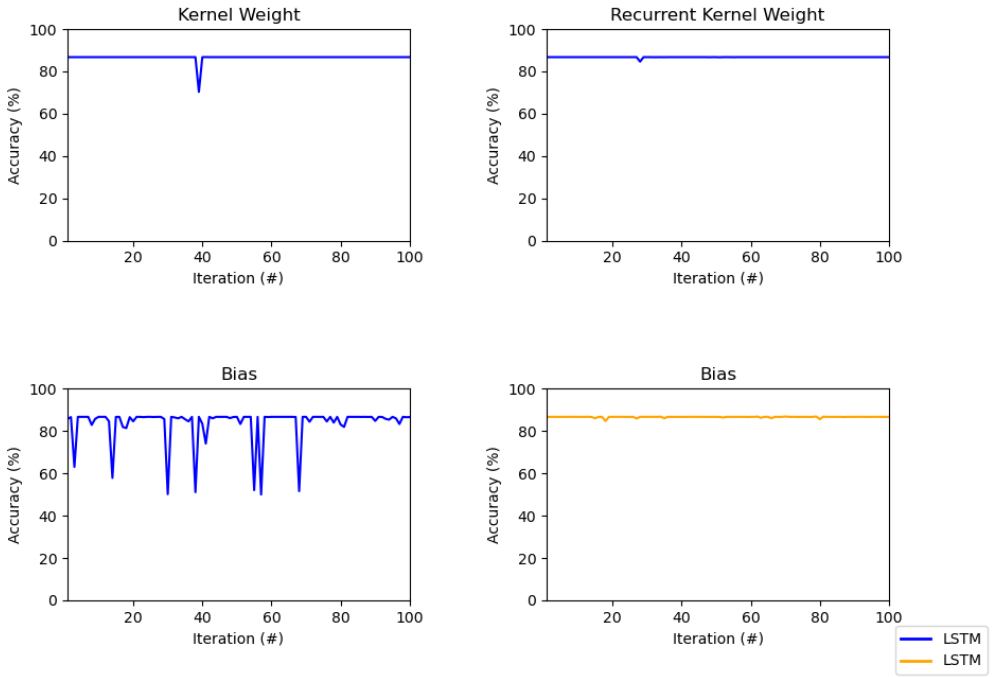


(d) Stacked Bidirectional LSTM Layers

Figure D.4. (cont'd) Modified LSTM Models with 1 SEU in Exponent Bit Field of Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

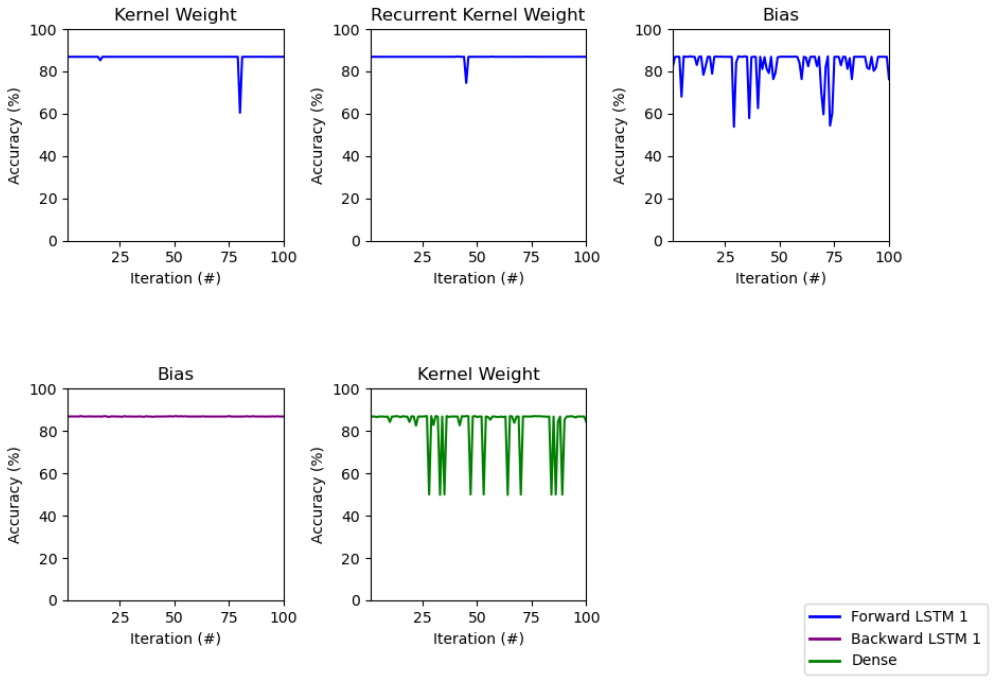


(a) LSTM Layer

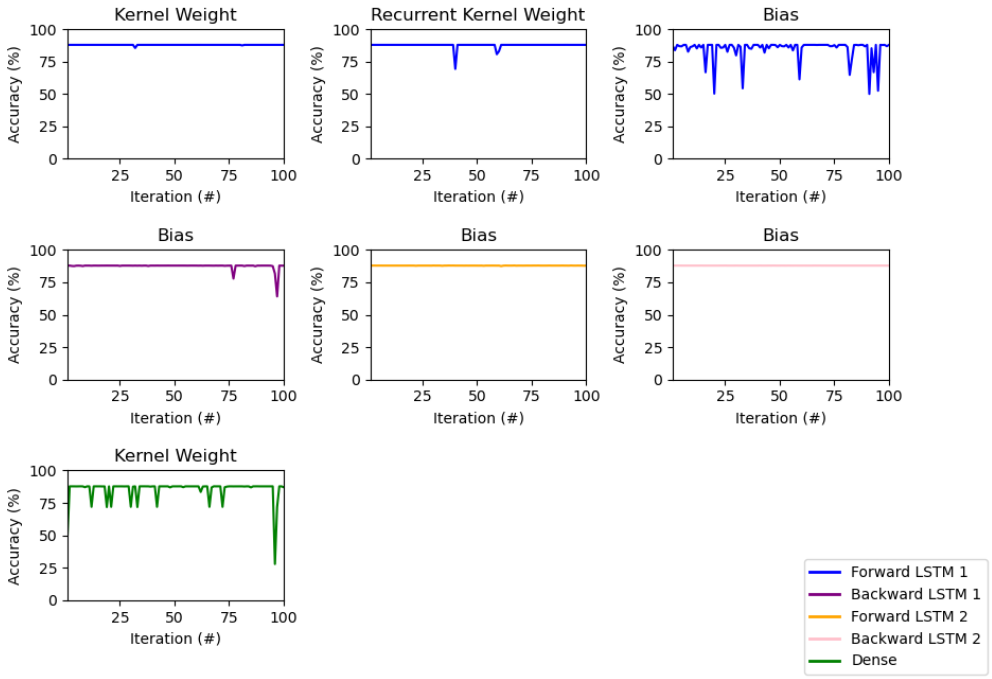


(b) Stacked LSTM Layers

Figure D.5. Modified LSTM Models with 10 SEU in Exponent Bit Field of Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

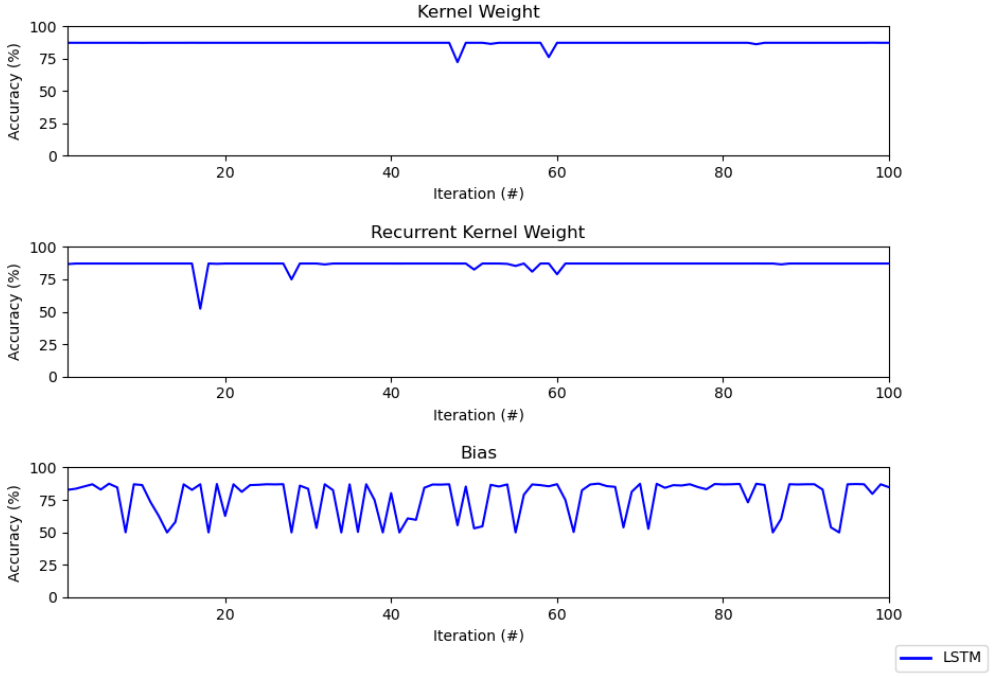


(c) Bidirectional LSTM Layer

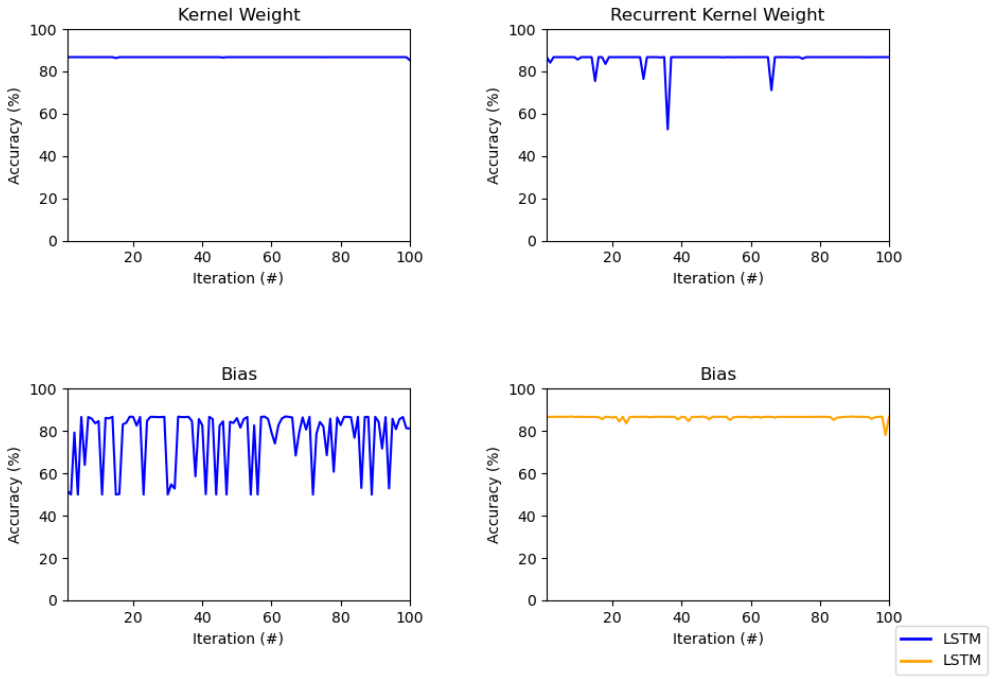


(d) Stacked Bidirectional LSTM Layers

Figure D.5. (cont'd) Modified LSTM Models with 10 SEU in Exponent Bit Field of Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

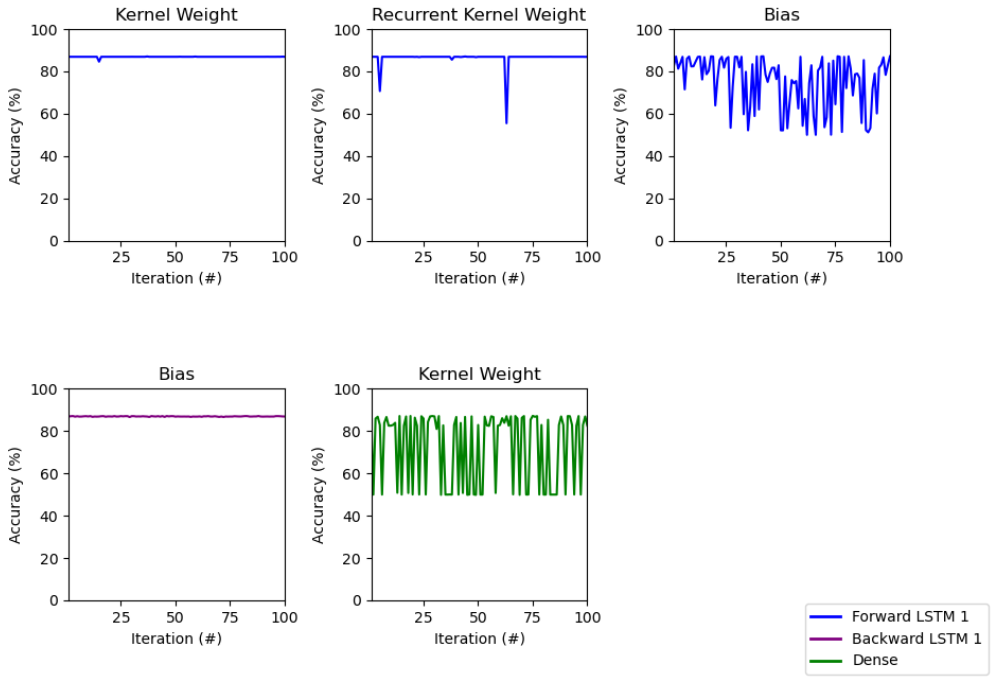


(a) LSTM Layer

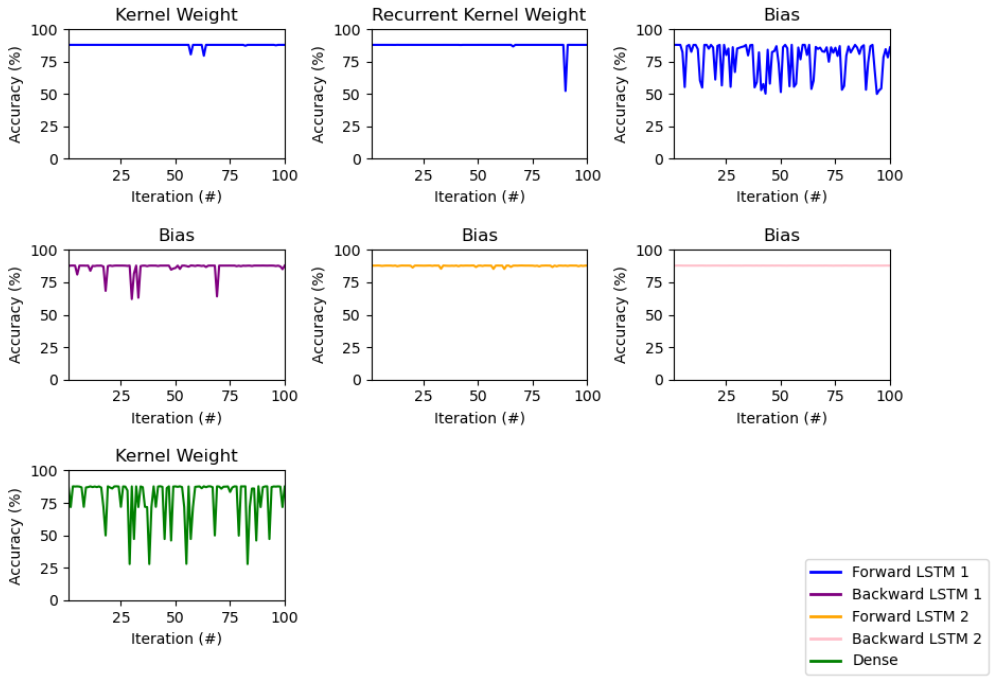


(b) Stacked LSTM Layers

Figure D.6. Modified LSTM Models with 20 SEU in Exponent Bit Field of Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations



(c) Bidirectional LSTM Layer



(d) Stacked Bidirectional LSTM Layers

Figure D.6. (cont'd) Modified LSTM Models with 20 SEU in Exponent Bit Field of Each Susceptible Set of Weights/Biases in Each Layer Over 100 Iterations

Appendix E. Hamming Encoding & Decoding Function Implementations

```
def hamming_encode_weights(weights):

num_pars = 3

# Flatten the weight tensor
weights_flat = tf.identity(weights)
weights_flat = tf.keras.backend.flatten(weights_flat)
weights_flat = tf.unstack(weights_flat)

# Move the exponent bits to the right
weights_int = tf.bitcast(weights_flat, tf.int32)
weights_shifted = tf.bitwise.right_shift(weights_int, 23)

# Convert the exponent bits to binary and split them into sets of 4 bits each
exponent = tf.reverse(tf.math.floormod(tf.bitwise.right_shift(tf.expand_dims(weights_shifted,1), tf.range(8)), 2), axis=[-1])
exponent = tf.reshape(exponent, [tf.shape(exponent)[0], 2, 4])

# Generator Matrix for 4 data bits
G = tf.constant(np.array([[1,1,0],
                          [1,0,1],
                          [0,1,1],
                          [1,1,1],]), dtype=tf.uint32)

# Generate the Parity bits for two sets of 4 data bits
parityBits = tf.matmul(exponent, tf.cast(G,tf.int32)) % 2

# Convert the parity bits from binary to decimal
parityBits = tf.reshape(parityBits, [tf.shape(parityBits)[0], 2*num_pars])
parityBits =  tf.bitcast(tf.reduce_sum(tf.reverse(parityBits, axis=[-1]) * 2 ** tf.range(2*num_pars), axis=[-1], keepdims=True), tf.int32)

# Flatten the parity bit tensor and add parity bits to weight
parityBits = tf.reshape(parityBits, [tf.shape(parityBits)[0]])
protectedWeights = tf.bitwise.bitwise_or(tf.bitwise.left_shift(tf.bitwise.right_shift(weights_int, 2*num_pars), 2*num_pars), parityBits)

# Format the protected weights to the original weight's dimensions
protectedWeights = tf.reshape(protectedWeights, weights.shape)

return tf.bitcast(protectedWeights, tf.float32)

```

Figure E.1. Code for Hamming Encoder for Fault Mitigation

```
def hamming_decode_weights(encoded_weights):

num_pars = 3
weights_int = tf.bitcast(encoded_weights, tf.int32)

# Flatten the weight tensor
weights_flat = tf.identity(weights_int)
weights_flat = tf.keras.backend.flatten(weights_flat)
weights_flat = tf.unstack(weights_flat)

# Convert the weights to binary and split them into sets of 4 bits each
parityBits = tf.reverse(tf.math.floormod(tf.bitwise.right_shift(tf.
    expand_dims(weights_flat,1), tf.range(num_pars*2)), 2), axis=[-1])
weights_shifted = tf.bitwise.right_shift(weights_flat, 23)
exponentBits = tf.reverse(tf.math.floormod(tf.bitwise.right_shift(tf.
    expand_dims(weights_shifted,1), tf.range(8)), 2), axis=[-1])

# Split exponents and parity bits into two groups
exponentBits = tf.reshape(exponentBits, [tf.shape(exponentBits)[0], 2,
    4])
parityBits = tf.reshape(parityBits, [tf.shape(parityBits)[0], 2,
    num_pars])
codewords = tf.concat([tf.gather(parityBits, [0,1], axis=-1),
                       tf.gather(exponentBits, [0], axis=-1)], -1)
codewords = tf.concat([codewords,
                       tf.gather(parityBits, [2], axis=-1)], -1)
codewords = tf.concat([codewords,
                       tf.gather(exponentBits, [1,2,3], axis=-1)], -1)

# Parity Check Matrix for 4 data bits (Trasposed & Flipped)
H_ti = tf.constant([[0,0,1],
                    [0,1,0],
                    [0,1,1],
                    [1,0,0],
                    [1,0,1],
                    [1,1,0],
                    [1,1,1]], dtype=tf.int32)

# Decode the codewords (We subtract from 7 because the identifier for
    errors goes left to right so we invert it)
decoded = tf.matmul(codewords, H_ti) % 2
decoded = tf.bitcast(tf.reduce_sum(tf.reverse(decoded, axis=[-1]) * 2 **
    tf.range(num_pars), axis=[-1], keepdims=True), tf.int32)
decoded = tf.reshape(decoded, [tf.shape(decoded)[0], 2])

# Convert the decoded location of errors to a mask where the 1 is in the
    error position
decoded = tf.where(decoded != 0, tf.pow(2, decoded), decoded)
decoded = tf.bitwise.right_shift(decoded, 1)

# Reverse the codewords so that most signfigant bit is on the left hand
    size (i.e. bit 7)
codewords = tf.bitcast(tf.reduce_sum(codewords * 2 ** tf.range(7), axis
```

```
    =[-1]), tf.int32)

# Use mask the codewords with the error position masks to correct any errors
fixed_codewords = tf.bitwise.bitwise_xor(codewords, decoded)
fixed_codewords = tf.identity(fixed_codewords)
fixed_codewords = tf.keras.backend.flatten(fixed_codewords)
fixed_codewords = tf.unstack(fixed_codewords)

# Convert the codewords to binary
fixed_codewords = tf.math.floormod(tf.bitwise.right_shift(tf.expand_dims(fixed_codewords,1), tf.range(7)), 2)
fixed_codewords = tf.reshape(fixed_codewords, [tf.shape(fixed_codewords)[0]/2, 2, 7])
parityBits = tf.gather(fixed_codewords, [0,1,3], axis=-1)
exponentBits = tf.gather(fixed_codewords, [2,4,5,6], axis=-1)
parityBits = tf.reshape(parityBits, [tf.shape(parityBits)[0], 6])
exponentBits = tf.reshape(exponentBits, [tf.shape(exponentBits)[0], 8])

# Convert the bits to decimal
parityBits = tf.bitcast(tf.reduce_sum(tf.reverse(parityBits,axis=[-1]) * 2 ** tf.range(6), axis=[-1]), tf.int32)
exponentBits = tf.bitcast(tf.reduce_sum(tf.reverse(exponentBits,axis=[-1]) * 2 ** tf.range(8), axis=[-1]), tf.int32)

#Move Exponent bits to the proper position
exponentBits = tf.bitwise.left_shift(exponentBits, 23)
weights_masked = tf.bitwise.bitwise_and(weights_flat, 2155872192)
weights_masked = tf.bitwise.bitwise_or(weights_masked, exponentBits)
weights_masked = tf.bitwise.bitwise_or(weights_masked, parityBits)
weights_masked = tf.reshape(weights_masked, tf.shape(encoded_weights))
weights_masked = tf.bitcast(weights_masked, tf.float32)

return weights_masked

```

Figure E.2. Code for Hamming Decoder for Fault Mitigation

References

[1] Martıén Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *Osdi*. Vol. 16. 2016. Savannah, GA, USA. 2016, pp. 265–283.

[2] Sui Chen et al. "Soft Error Resilience in Big Data Kernels through Modular Analysis". In: *J. Supercomput.* 72.4 (Apr. 2016), pp. 1570–1596. ISSN: 0920-8542. DOI: 10.1007/s11227-016-1682-2. URL: https://doi.org/10.1007/s11227-016-1682-2.

[3] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. "Ranger: Boosting Error Resilience of Deep Neural Networks through Range Restriction". In: *CoRR* abs/2003.13874 (2020). arXiv: 2003.13874. URL: https://arxiv.org/abs/2003.13874.

[4] Zitao Chen et al. "BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356177. URL: https://doi.org/10.1145/3295500.3356177.

[5] Zitao Chen et al. "TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications". In: *CoRR* abs/2004.01743 (2020). arXiv: 2004.01743. URL: https://arxiv.org/abs/2004.01743.

[6] Li Deng and Dong Yu. *Deep Learning: Methods and Applications*. 2014.

[7] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[8] Lide Duan, Bin Li, and Lu Peng. "Versatile prediction and fast estimation of Architectural Vulnerability Factor from processor performance metrics". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 129–140. DOI: 10.1109/HPCA.2009.4798244.

[9] Lide Duan et al. "Comprehensive and Efficient Design Parameter Selection for Soft Error Resilient Processors via Universal Rules". In: *IEEE Transactions on Computers* 63.9 (2014), pp. 2201–2214. DOI: 10.1109/TC.2013.24.

[10] E. Dubrova. *Fault-Tolerant Design*. Springer New York, 2013. ISBN: 9781461421139. URL: https://books.google.com/books?id=FRs%5C_AAAAQBAJ.

[11] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[13] Barry W. Johnson. "Fault-Tolerant Microprocessor-Based Systems". In: *IEEE Micro* 4.6 (1984), pp. 6–21. DOI: 10.1109/MM.1984.291277.

[14] Travis LeCompte et al. "Protecting Synchronization Mechanisms of Parallel Big Data Kernels via Logging". In: *IEEE Transactions on Computers* 71.9 (2022), pp. 2156–2162. DOI: 10.1109/TC.2021.3122993.

[15] Travis LeCompte et al. "Soft error resilience of Big Data kernels through algorithmic approaches". In: *The Journal of Supercomputing* 73 (Nov. 2017). DOI: 10.1007/s11227-017-2042-6.

[16] Guanpeng Li et al. "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications". In: *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–12.

[17] Andrew L. Maas et al. "Learning Word Vectors for Sentiment Analysis". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. URL: http://www.aclweb.org/anthology/P11-1015.

[18] Elaheh Malekzadeh et al. "The Impact of Faults on DNNs: A Case Study". In: *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2021, pp. 1–6. DOI: 10.1109/DFT52944.2021.9568340.

[19] Christopher Olah. *Understanding LSTM Networks*. Aug. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[20] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[21] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. "Efficient On-Line Error Detection and Mitigation for Deep Neural Network Accelerators". In: *Computer Safety, Reliability, and Security*. Ed. by Barbara Gallina, Amund Skavhaug, and Friedemann Bitsch. Cham: Springer International Publishing, 2018, pp. 205–219. ISBN: 978-3-319-99130-6.

[22] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[23] Zheyu Yan et al. “When Single Event Upset Meets Deep Neural Networks: Observations, Explorations, and Remedies”. In: *CoRR* abs/1909.04697 (2019). arXiv: 1909.04697. URL: http://arxiv.org/abs/1909.04697.

[24] James Ziegler. “Terrestrial cosmic rays”. In: *IBM Journal of Research and Development* 40 (Jan. 1996), pp. 19–40. DOI: 10.1147/rd.401.0019.

Vita

Christopher P. Vasquez was born in Metairie, LA, USA in 2000. Studying at Louisiana State University (LSU) in Baton Rouge, he obtained a bachelor's in computer engineering and a bachelor's in computer science. Currently, he is pursuing his master's degree in electrical engineering at LSU and plans to graduate in August 2023.

During his time at LSU, he has worked on various projects including an AR project for composite material manufacturing, a medical device for Runatek, coastal model software funded by the National Science Foundation, and a contact tracing mobile application funded by the National Institute of Health. In addition to general software development, his expertise lies primarily in deep learning and software acceleration methods through parallelization.