Simulating Single Event Upsets with Bit-Level Fault Injection in a Neural Network

Center for Reliable and Trusted Electronics (CREATE)

E. Azziz, *T. Peyton, M.E. Loveless Ph.D., T.D. Loveless Ph.D.*
Intelligent Systems Engineering, Center for Reliable & Trusted Electronics, Indiana University, Bloomington, IN



Introduction

It has been shown that electronic systems are susceptible to radiation effects present in harsh environments such as space, giving rise to the study of **Single-Event Effects (SEEs)**. As more intelligent systems that include *machine learning* and *artificial intelligence* are deployed within radiation-prone electronic systems, understanding the effects of radiation becomes critical.

This project investigates the way a specific type of **SEE** known as a **Single-Event Upset (SEUs)** affects neural network models.

- **SEUs** can affect these models by *flipping bit values* within the stored weights, biases, and other components of a deep neural network.

In order to study this effect, a **fully connected neural network** was trained on the **MNIST dataset**, reaching **97.2%** baseline accuracy.
- With this architecture:
  - **Single-bit flips** are incrementally injected across all bits of each **32-bit float weight** in selected dense layers.
  - For each weight:
    - A single bit is flipped
    - The model is evaluated on a **1,000-sample** test subset
    - Original value is *restored* before moving onto next bit.

This cycle is repeated for **each individual bit** in **each individual FP32 weight**, creating **millions** of inferences per model.
This process enables analysis of how *individual* bit flips impact model performance.

Objective

Develop a fault injection framework to simulate single-bit errors in neural network weights and evaluate their impact on model accuracy.

References

[1] Y. LeCun, L. Bottou et al. “The MNIST Dataset Of Handwritten Digits (Images)”
[2] T. Peyton, J. Carpenter et al, “Classification of Microelectronics Radiation Effects Using Unsupervised Machine Learning.”
[3] G. Li, K. Pattabiraman et al, “TensorFI: A Configurable Fault Injector for TensorFlow Applications.”
[4] A4RES Symposium 2025.

Acknowledgments

Special thanks to S. Lasassmeh Ph.D, A Buechelein, H. Amac, for mentorship and IU CREATE and SCALE for funding this project..

Computational resources were provided by the Indiana University Bloomington High Throughput Computing (BHTC). This work utilized *Quartz*, Indiana University’s high-throughput computing cluster.

Model Architecture

The MNIST dataset is a collection of **70k** images containing *handwritten* digits (0-9). The purpose of this neural network is to *analyze* handwritten digits then **accurately classify** each image according to the digit it represents.

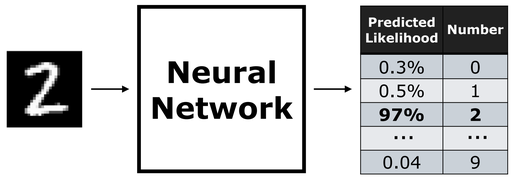


Fig. 1. MNIST Architecture. [1]

Each MNIST image is originally a *28×28* pixel array. These arrays are **flattened** into a *784×1* vector so they can be passed into a dense layer, which requires **1-dimensional** inputs.

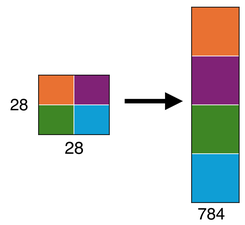


Fig. 2. Flattening of array.

A TensorFlow model was defined with:
`tf.keras.Sequential([])`
`tf.keras.layers.Input(shape=(784,)),`
`tf.keras.layers.Dense(128),`
`tf.keras.layers.Dense(64)`, and
`tf.keras.layers.Dense(10)`.

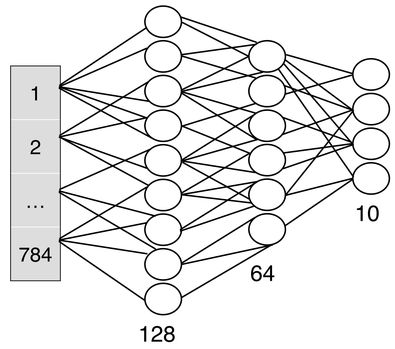


Fig. 3. Neural Network Architecture

During training, a neural network **learns** by adjusting its “*weights*”. These weights are essentially the “memory” or “thinking” of the model. In our network, for example, the weights connecting Dense Layer 1 to Dense Layer 2 are stored in a [128, 64] tensor, which represents **all the relationships** between those two layers.

Bit-Fault Injection

Individual weights and biases are stored in *32-bit fields*, though this can be quantized. To simulate a SEU, a **single bit** is *flipped* within the 32-bit field.

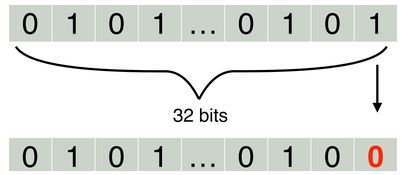


Fig.4. Example of Single Bit flip

Bit-flip injection begins at weight [0,0], *sequentially* flipping bits 1 through 32. After each flip, the model is **evaluated**, accuracy is **logged**, and the original value is **restored** *before* the next flip. This process *repeats* for every bit in every weight: from **[0,0]** to the f**inal weight** in the layer.

For a layer with 128×64 weights, this results in 128×64×32 = 262,144 total evaluations. This same bit-flipping procedure is then applied to the *subsequent layer*.

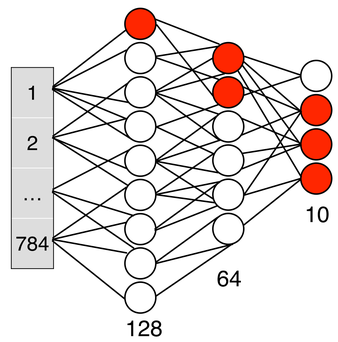


Fig.5. Neural Network Error Propagation

If a numerical fault is found in the weights of a model, this creates **errors** that *propagate* down to model output & can *sway predictions*. This speaks to how *dangerous* SEUs can be, as a **single upset** propagates down the entire model up until output.

```
for i in range(w.shape[0]):
    for j in range(w.shape[1]):
        orig_val = w[i, j].numpy()
        for bit_pos in range(32):
            # flip
            new_val = flip_bit(orig_val, bit_pos)
            w.assign(tf.tensor_scatter_nd_update(w, [[i, j]], [new_val]))

            # evaluate
            _, acc = model.evaluate(sample_x, sample_y, verbose=0)
            acc_drop = float(base_acc - acc)

            # restore
            w.assign(tf.tensor_scatter_nd_update(w, [[i, j]], [orig_val]))
```

Fig.6. Block of Code That Performs Flips Throughout a Layer

The code block above performs bit flips, throughout an entire layers weights.

Results

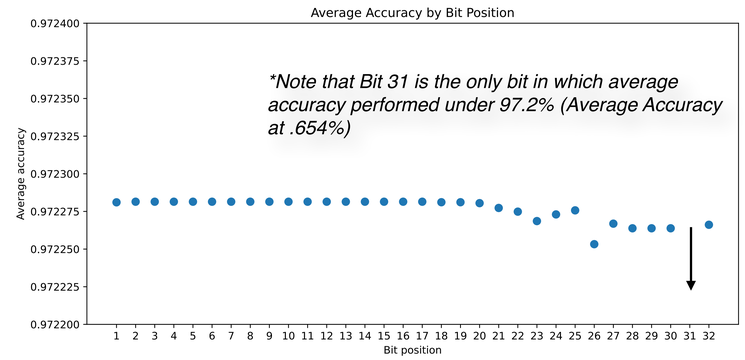


Fig.7. Average Accuracy by Bit Position

Baseline accuracy: 97.2%

Bits 1–20: No changes in accuracy; suggests **general resilience** to SEUs in low-order bits

Bits 21–30 & Bit 32: Minor changes observed, all within **1e–5**, indicating **negligible impact**

Bit 31:

- 24.8% drop (Layer 1), 32.0% drop (Layer 2).
- Indicates **critical weakness**

Inter-layer differences:

- 8% average difference in accuracy drop between layers 1 & 2 at bit 31
- Further testing needed to determine how much **layer architecture** influences fault sensitivity
- *Note: The layer-wise differences are not shown in the figure due to space constraints but were observed during testing*

Total evaluations run: 282,624

Discussion & Future Work

- **Bit position 31** showed the most vulnerability to SEUs, shown by large accuracy drops. This is likely due to its role in the **exponent field** of 32-bit floats.
  - Strategies for mitigation include **model quantization**, which *reduces the number of bits* used to represent weights. This technique *may* potentially lower SEU vulnerability.
  - An **8% accuracy difference** between layers at this bit position suggests that faults closer to the output layer *may* be more damaging. This trend requires validation across *other model architectures* to confirm.
- These findings are **specific** to the architecture of the model tested; they **do not** generalize to all neural networks.
  - This procedure must be tested with a variety of different neural networks to understand how model *architecture* affects vulnerability
- **Computational Cost & Optimization**
  - Performing 32 different evaluations per weight is *computationally expensive*.
  - To address this, we plan to implement **parallel processing** using *multiple GPUs*.
  - As opposed to sequentially flipping bits, each GPU will handle *a distinct subset* of the weights. This will greatly **reduce total runtime** & will **enable larger-scale testing**, especially for more sophisticated models.
  - *Note: This multiprocessing workflow is under testing with simpler, one-layer models & is currently being implemented within the MNIST models architecture.*

At scale, this bit-flipping procedure is intended for use with more *sophisticated neural networks*, such as **ImageNet** based architecture, as those are more representative of what’s deployed in real world **radiation-prone environments**. This enables *analysis* of how different neural network *architecture* affects it’s **susceptibility to SEUs**. While transferring the fault injection logic from this models architecture to a larger models is straightforward, difficulty arises in *computational cost* as number of evaluations turns from **hundreds of thousands** per model to t**ens of millions**. This speaks to the importance of developing multiprocessing procedures.