CS6886 Systems Engineering for DL Project Report

CS25D007 Rahul Anilkumar

RePHINO - Part II - Reliability Analysis of Fourier Neural Operators

1 Introduction

Neural Operators (NO) enable scientific computations by learning a mapping between infinite-dimensional function spaces to solve partial differential equations (PDEs) quickly. Fourier Neural Operator is a type of Neural operator that learns the mapping between functions by performing calculations in the Fourier Space. It consists of 3 major layers:

- Lifting Layers: Lifts the input data from low-dimensional physical space to a high-dimensional latent space.
- Spectral Convolution Layers (Fourier Layer): Responsible for the complete global operation: it applies an FFT (to move to the frequency domain), performs the learned convolution in frequency space, and then applies an Inverse-FFT (to return to the spatial domain).
- Projection Layers: project the high-dimensional latent data back to the desired output/solution space.

Motivation: FNOs can enable scientific computation with relatively simpler compute once trained, which can enable a lot of potential in edge applications. Deploying them in radiation-prone environments like space, however, would require additional guarantees to ensure their reliable operation.

In Section 2, we look at analysing the Fault Tolerant capabilities of different layers of an FNO to Single Event Upsets caused by radiation impact, by simulating bit-flips in the weights and activations of the model. It includes detailed discussions on Methodology followed in defining fault regimes and mapping existing studies to FNOs. Section 3 illustrates the Results from the Fault Injection Campaigns, and presents their Analysis. Section 4 proposes two techniques to improve the SEU Fault tolerance of FNOs without significant resource usage. Section 5 covers the implementation of FNO on an edge device (Raspberry Pi). Section 6 closes the study with future directions the research can take.

2 Reliability Analysis through Fault Injection Campaigns

Electronic devices operating in the harsh environments of space is prone to radiation induced upsets, which can introduce functional errors, or even corrupt the entire system. Designers often

take care to strengthen their designs against such upsets with techniques such as redundancy or error correction codes. Such techniques, however, applied at the system level can consume a lot of chip area and power.
To overcome such overheads, studies have been performed to identify the most critical areas that need protection [1], and/or devising hardware [2] and software [3] techniques to arrive at an optimized solution. Most studies have been done on processors, since they have been a crucial part of space programs for decades.
In this section, we will briefly look at the physical phenomenon involved with radiation upsets, how they are modeled in conventional literature, and how they can be mapped to FNOs.

2.1 Radiation Induced Upsets

Radiation-induced upsets are caused due to the impact of ionizining particles causing the state of electronic circuits to change. The most impactful and studied of them is the Single Event Effect (SEE), which manifests itself in three ways [4]:

1. Single Event Upset (SEU): SEUs occur when an incident particle deposits enough charge (known as critical charge) on a storage element that results in the stored value getting flipped.

2. Single Event Transient (SET): SETs are transient voltage pulses that propagate through combinational elements, and are only of concern if they get captured into a storage element.

3. Single Event Latchup (SEL): SEL is the most dangerous of the three in that it can cause permanent damage to the device. SELs occur when the incident particles triggers a parasitic thyristor structure in the device, causing the current to increase exponentially till device burnout.

The solutions to these problems are as follows:

1. SEU: System-level techniques like Redundancy and Error Correction codes, circuit and technology level techniques.

2. SET: Circuit level techniques to filter large transients.

3. SEL: SEL can only be solved with device level techniques. Safeguard mechanisms like current limiting and power-cycling are used, however, architectural techniques have no role to play here.

In this project, we consider the impact of SEUs and SETs on the Fourier Neural Operator, and propose techniques to lessen the impact of SEUs.

2.2 Structure of Fourier Neural Operators

Fourier Neural Operators (FNO) are typically composed of three main parts: an input encoder (Lifting Layer), a series of stacked Fourier Blocks that form the processing core, and an output decoder (Projection Layer).

Lifting Layer (Encoder)

Purpose: To lift the input data from its low-dimensional physical space to a high-dimensional latent representation.

Implementation: A point-wise operation, typically a Multi-Layer Perceptron (MLP) or a 1×1 convolution, applied at every spatial/temporal location.

Example: Expands the input channels to the hidden dimension of the model.

Fourier Blocks (Processing Core)

Purpose: The primary processing unit of the FNO, typically stacked N times. Each block mixes information globally in the frequency domain and locally in the spatial domain.

Implementation: Each Fourier Block consists of two parallel paths, followed by a channel-mixing MLP:

- **Spectral Path:**
  - FFT: Transform data to the frequency domain.
  - Filtering: Multiply by learnable complex-valued weights (global convolution).
  - IFFT: Transform back to the spatial domain.
- **Spatial Path:** Residual skip connection, typically implemented as a 1×1 convolution.
- **Combination:** Add spectral and spatial outputs, then apply a non-linear activation.
- **Channel MLP:** Point-wise MLP to mix information across channels at each spatial location.

Projection Layer (Decoder)

Purpose: To project the high-dimensional latent data back to the desired output/solution space.

Implementation: Point-wise MLP or 1×1 convolution, similar to the lifting layer.

2.3 Methodology

In this subsection, we will first establish the mapping of SEUs and SETs to FNOs to analyse their impact, and define fault regimes based on them. We will also classify the extent of their impact to the overall output of the FNO.

2.3.1 SEU and SET in FNOs

1. SEU: Since SEU is a bit-flip whose impact is permanent until written over, Bit-flips in Weights can be considered as a manifestation of SEU.
2. SET: SETs are temporary, in that they do not affect further computations from the next cycle. Bit-flips in Activations can be considered as an SET on FNO.

2.3.2 Fault Regimes

The extent of faults can be estimated based on the number of bit-flips caused by an incident particle and the location of faults. Based on this, we establish the following Fault Regimes:

1. **Isolated Single Event Upsets**: Single bit flip in a layer of the FNO
2. **Clustered Multi-Bit Upsets**:
   (a) **Concentrated Multi-Bit Upsets**: Multiple bit-flips in the same layer
   (b) **Distributed Multi-Bit Upsets**: Multiple bit-flips across adjacent layers

2.3.3 Impact of Faults

The impact of faults can be broadly classified into two categories [5]:

1. **Silent Data Corruption (SDC)**: SDC arises when the inference completes without errors, but the final output is tainted by the bit-flip.
2. **Functional Interruption (FI)**: FI refers to a scenario when the inference is not able to complete and triggers some kind of interrupt or exception.

In case of FNOs, Functional Interruptions can be considered as the generation of a NaN (Not a Number) or Infinity due to the bit-flips, which would be quickly recognizable and make further processing difficult. Any other changes to output can be attributed to Silent Data Corruptions.

2.3.4 Conventional Methodology

Fault injection campaigns are performed at different levels for processors and neural networks. Processors are conventionally tested at RTL level [6], or even fabricated device/FPGA level, with hooks to inject faults and monitor the impact. Neural Networks are often evaluated at Simulation level [7], by injecting faults into the FP representations of weights and activations [8]-[9]. We will adopt a similar methodology for FNOs in this project.

2.3.5 Test Methodology for FNOs

Based on the methodology and fault regimes defined above, we run fault injection campaigns on the FNO with bit-flips in the weights and activations to simulate SEU and SETs. The perturbation caused due to the bit-flip is estimated using the change in Mean Squared Error (MSE) value against the output originally predicted by the FNO without any faults injected. Using this scheme, the fault campaigns are run for Isolated Single Event Upsets, Concentrated Multi-Bit Upsets and Distributed Multi-Bit Upsets, and the results are listed in the following section.

2.4 Fault Injection Campaigns in FNO

The baseline FNO model under consideration has the following configuration:

```
The model has 4469601 parameters.

### MODEL ###
FNO(
  (positional_embedding): GridEmbeddingND()
  (fno_blocks): FNOBlocks(
    (convs): ModuleList(
      (0-3): 4 x SpectralConv(
        (weight): DenseTensor(shape=torch.Size([32, 32, 32, 17]), rank=None)
      )
    )
    (fno_skips): ModuleList(
      (0-3): 4 x Flattened1dConv(
        (conv): Conv1d(32, 32, kernel_size=(1,), stride=(1,), bias=False)
      )
    )
    (channel_mlp): ModuleList(
      (0-3): 4 x ChannelMLP(
        (fcs): ModuleList(
          (0): Conv1d(32, 16, kernel_size=(1,), stride=(1,))
          (1): Conv1d(16, 32, kernel_size=(1,), stride=(1,))
        )
      )
    )
    (channel_mlp_skips): ModuleList(
      (0-3): 4 x SoftGating()
    )
  )
  (lifting): ChannelMLP(
    (fcs): ModuleList(
      (0): Conv1d(3, 64, kernel_size=(1,), stride=(1,))
      (1): Conv1d(64, 32, kernel_size=(1,), stride=(1,))
    )
  )
  (projection): ChannelMLP(
    (fcs): ModuleList(
      (0): Conv1d(32, 64, kernel_size=(1,), stride=(1,))
      (1): Conv1d(64, 1, kernel_size=(1,), stride=(1,))
    )
  )
)
```

The following layers are identified:

```
Layer 0   : fno_blocks.convs.0 [Real]
Layer 1   : fno_blocks.convs.0 [Imag]
```

```
Layer 2   : fno_blocks.convs.1 [Real]
Layer 3   : fno_blocks.convs.1 [Imag]
Layer 4   : fno_blocks.convs.2 [Real]
Layer 5   : fno_blocks.convs.2 [Imag]
Layer 6   : fno_blocks.convs.3 [Real]
Layer 7   : fno_blocks.convs.3 [Imag]
Layer 8   : fno_blocks.fno_skips.0.conv
Layer 9   : fno_blocks.fno_skips.1.conv
Layer 10  : fno_blocks.fno_skips.2.conv
Layer 11  : fno_blocks.fno_skips.3.conv
Layer 12  : fno_blocks.channel_mlp.0.fcs.0
Layer 13  : fno_blocks.channel_mlp.0.fcs.1
Layer 14  : fno_blocks.channel_mlp.1.fcs.0
Layer 15  : fno_blocks.channel_mlp.1.fcs.1
Layer 16  : fno_blocks.channel_mlp.2.fcs.0
Layer 17  : fno_blocks.channel_mlp.2.fcs.1
Layer 18  : fno_blocks.channel_mlp.3.fcs.0
Layer 19  : fno_blocks.channel_mlp.3.fcs.1
Layer 20  : lifting.fcs.0
Layer 21  : lifting.fcs.1
Layer 22  : projection.fcs.0
Layer 23  : projection.fcs.1
```

The Sloshing FNO is much larger, although follows a similar architecture:

```
The model has 18887843 parameters.

### MODEL ###
 FNO(
  (positional_embedding): GridEmbeddingND()
  (fno_blocks): FNOBlocks(
    (convs): ModuleList(
      (0-3): 4 x SpectralConv(
        (weight): DenseTensor(shape=torch.Size([32, 32, 16, 16, 9]), rank=None)
      )
    )
    (fno_skips): ModuleList(
      (0-3): 4 x Flattened1dConv(
        (conv): Conv1d(32, 32, kernel_size=(1,), stride=(1,), bias=False)
      )
    )
    (channel_mlp): ModuleList(
      (0-3): 4 x ChannelMLP(
        (fcs): ModuleList(
          (0): Conv1d(32, 16, kernel_size=(1,), stride=(1,))
          (1): Conv1d(16, 32, kernel_size=(1,), stride=(1,))
```

```
        )
      )
    )
    (channel_mlp_skips): ModuleList(
      (0-3): 4 x SoftGating()
    )
  )
  (lifting): ChannelMLP(
    (fcs): ModuleList(
      (0): Conv1d(6, 64, kernel_size=(1,), stride=(1,))
      (1): Conv1d(64, 32, kernel_size=(1,), stride=(1,))
    )
  )
  (projection): ChannelMLP(
    (fcs): ModuleList(
      (0): Conv1d(32, 64, kernel_size=(1,), stride=(1,))
      (1): Conv1d(64, 3, kernel_size=(1,), stride=(1,))
    )
  )
)
```

The Spectral Conv layers have imaginary weights, and are stored in Pytorch as complex values (2 float32 values - real and complex parts), are are hence counted as 2 layers each.
The three Fault Injection campaigns are established as below:

2.4.1 Isolated Fault Injection

The objective of this regime is to assess the layer-wise vulnerability to a single bit upset.
Weight Fault Injection: One randomly selected weight element per target layer is perturbed by flipping a single bit within one of four prescribed bit quadrants:

- Q1: bits 0–7 (lower mantissa)
- Q2: bits 8–15 (middle mantissa)
- Q3: bits 16–23 (upper mantissa)
- Q4: bits 24–31 (exponent + sign)

For spectral convolution layers, real and imaginary components are evaluated independently.
Activation Fault Injection: One randomly selected activation in the layer output tensor is perturbed using the same quadrant policy.
Procedure:

- For each layer × quadrant, `NUM_TRIALS` (in this case 100) independent injections are performed with fixed random seeds.
- For each trial, the model processes `TEST_SAMPLES` inputs (in this case 50) ; outputs are compared to fault-free baselines to compute MSE.

- Any NaN or Inf in output tensors is recorded as a functional interrupt (FI).

From these, the per-layer distribution of changes to MSE are computed to estimate a particular layer's sensitivity to an isolated bit-flip.

2.4.2 Clustered Fault Injections

The objective of this regimes is to simulate multi-bit upsets (MBUs) in the same layer.
Procedure: For each selected layer, K flips are applied in the same layer. A maximum number of 5 bit-flips are injected based on in-flight information from [10].

2.4.3 Distributed Fault Injections

The objective of this regimes is to simulate single-bit upsets in adjacent layers.
Procedure: For each selected layer, a single-bit flip is introduced into that layer, followed by a 60% probability that a bit-flip is introduced in the next layer (total 2 bit-flips), a 20% chance for another layer (total 3), 10% chance each for 4 and 5 bit-flips.

3 Results

Following are the results from the fault injection campaigns run on the FNO (the first set of results are from baseline Darcy Flow FNO, Sloshing FNO are added in Appendix):

3.1 Isolated Single Event Upset

3.1.1 Silent Data Corruptions (SDCs)

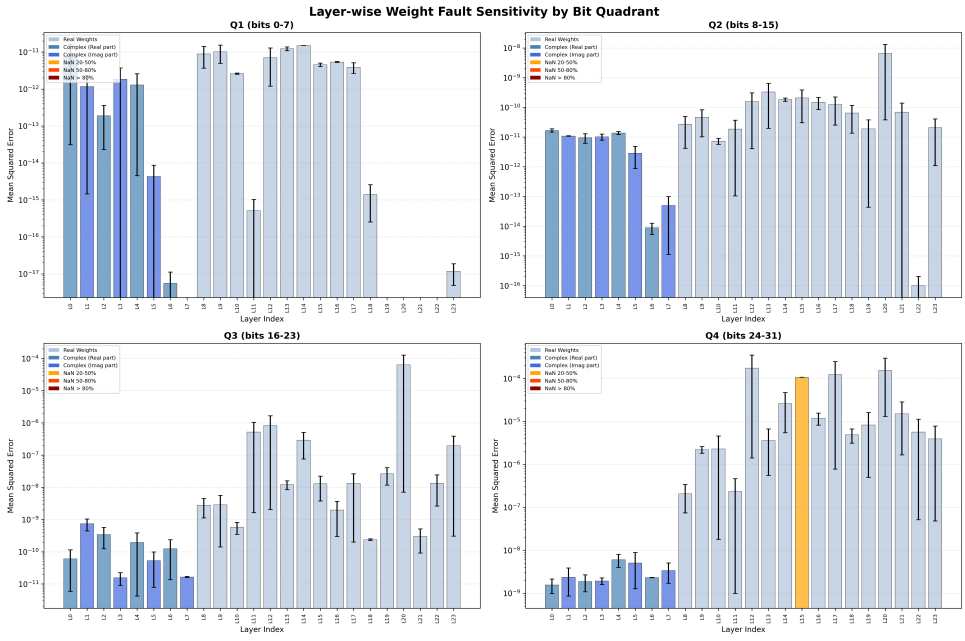

Figure 1: Layer-wise sensitivity to single-bit upset in Weights (Bit Quadrant-wise)

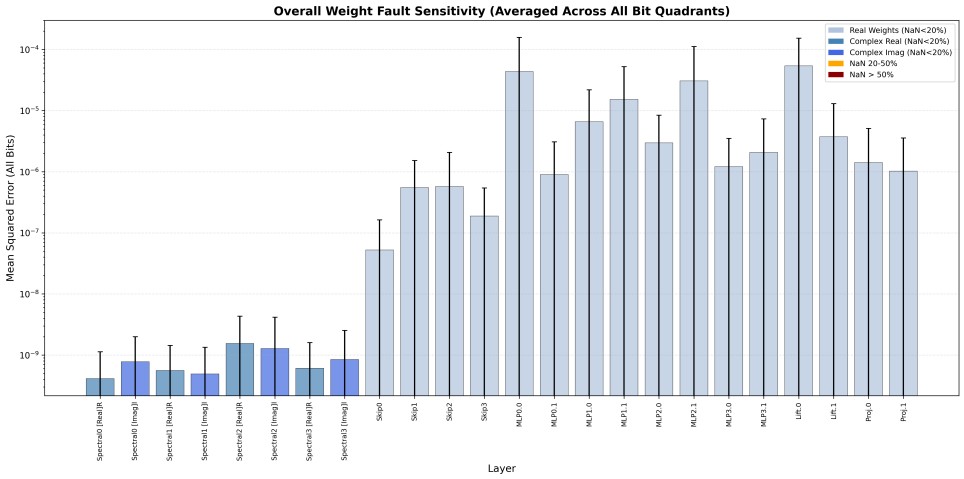

Figure 2: Layer-wise sensitivity to single-bit upset in Weights (Overall)

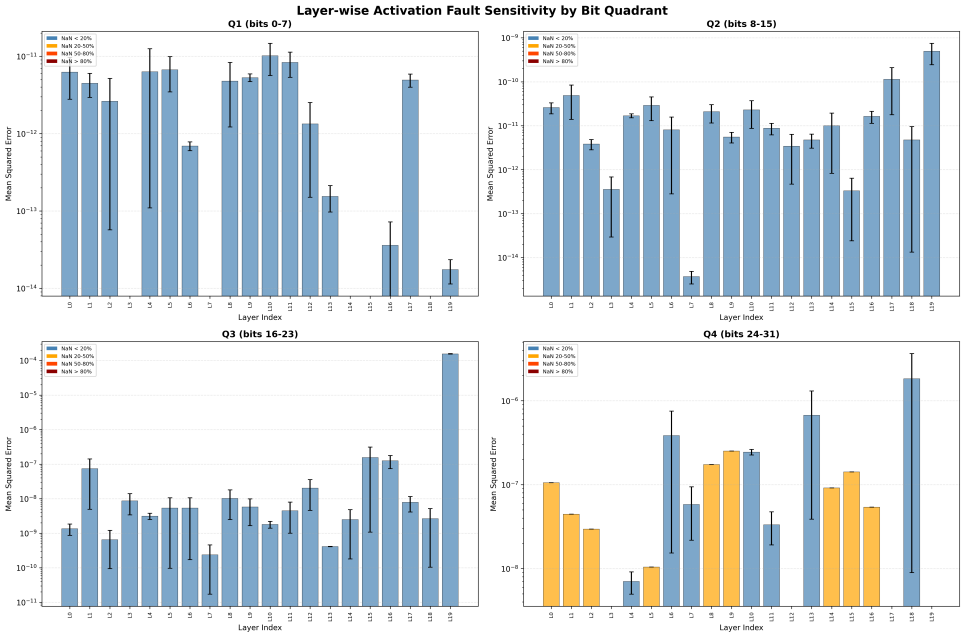

Figure 3: Layer-wise sensitivity to single-bit upset in Activations (Bit Quadrant-wise)

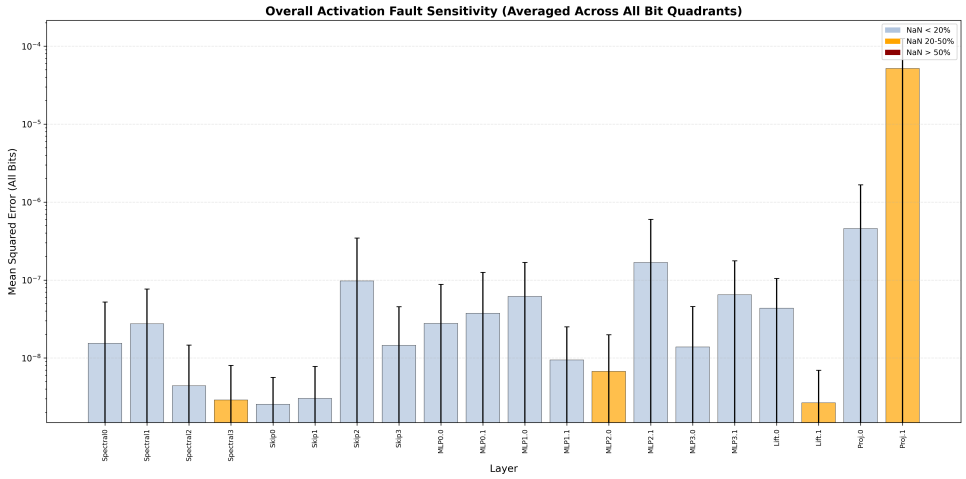

Figure 4: Layer-wise sensitivity to single-bit upset in Activations (Overall)

3.1.2 Functional Interruptions (FIs)

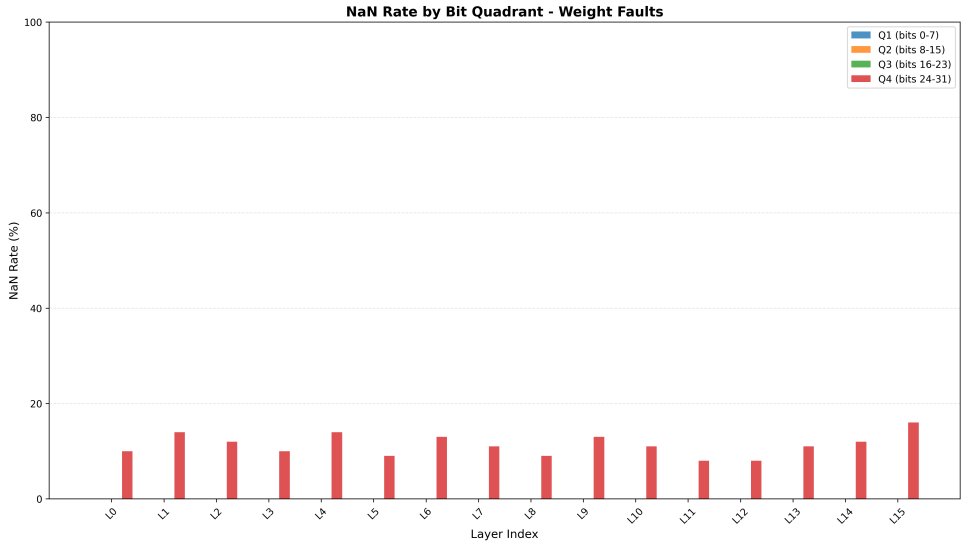

Figure 5: Functional Interruptions due to fault injection in Weights

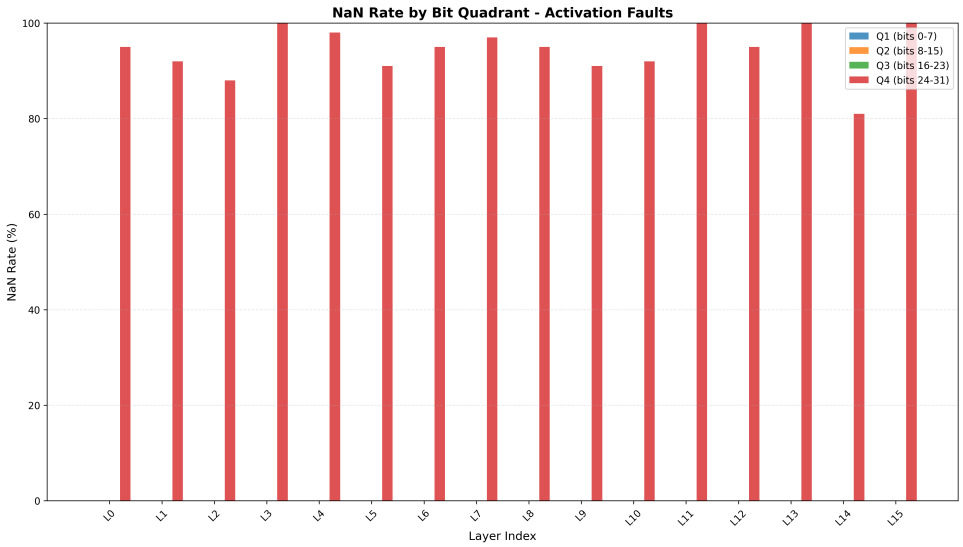

Figure 6: Functional Interruptions due to fault injection in Activations

3.2 Clustered Multi-Bit Upset

3.2.1 Concentrated Multi-Bit Upset

Silent Data Corruptions (SDCs)

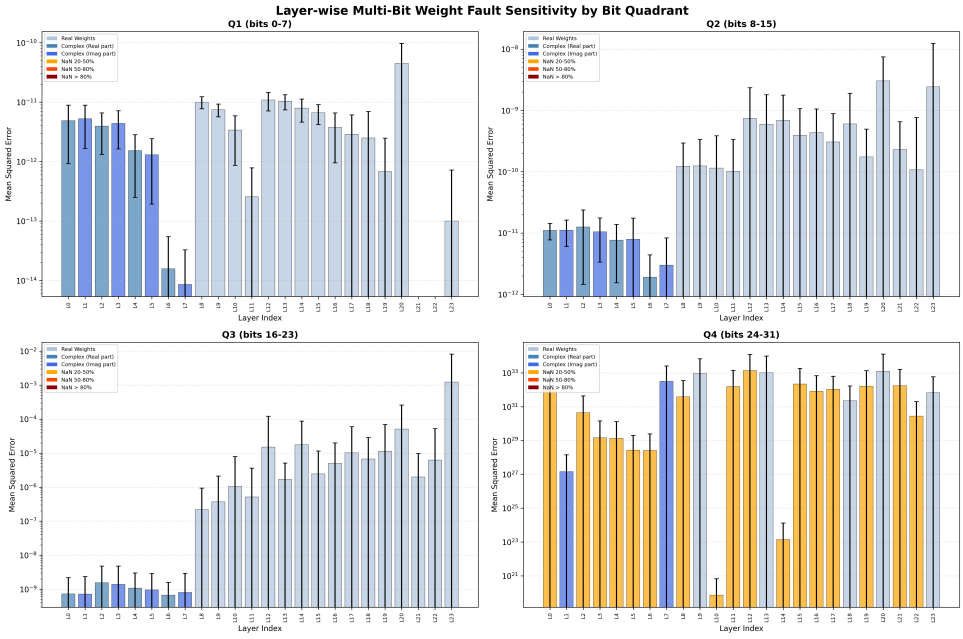

Figure 7: Layer-wise sensitivity to multi-bit upset in the same layer for Weights (Bit Quadrant-wise)

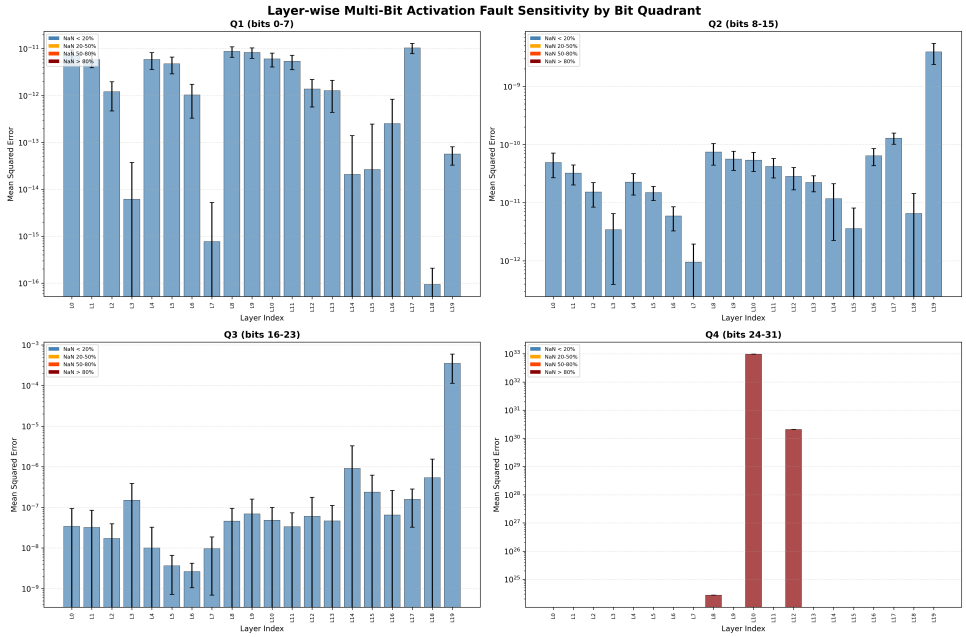

Figure 8: Layer-wise sensitivity to multi-bit upset in the same layer for Activations (Bit Quadrant-wise)

Functional Interruptions (FIs)

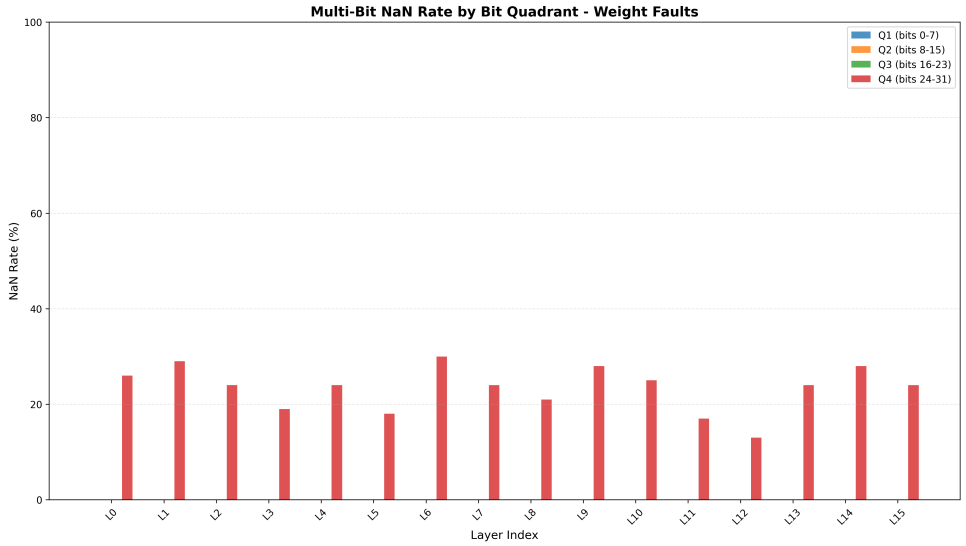

Figure 9: Functional Interruptions due to multi-bit fault injections in Weights

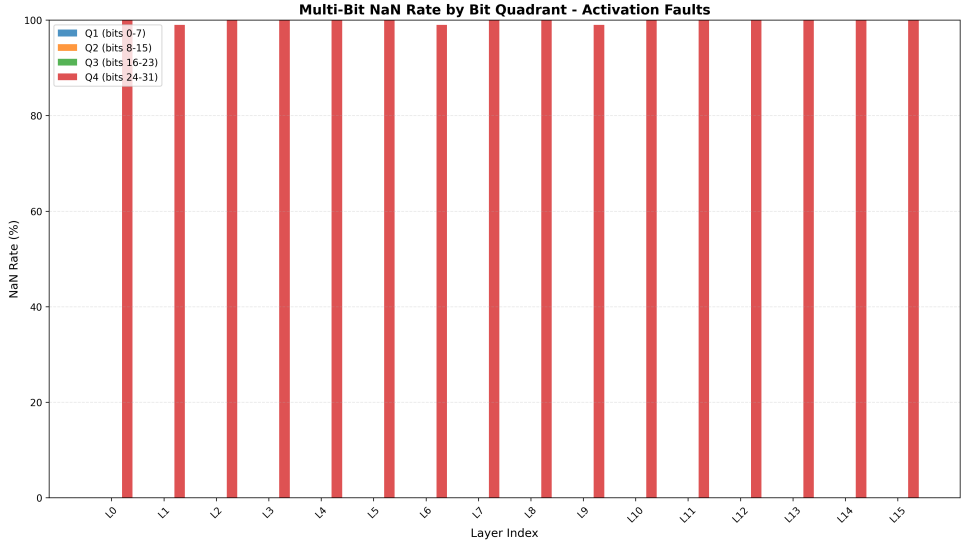

Figure 10: Functional Interruptions due to multi-bit fault injections in Activations

3.2.2 Distributed Multi-Bit Upset

Silent Data Corruptions (SDCs)

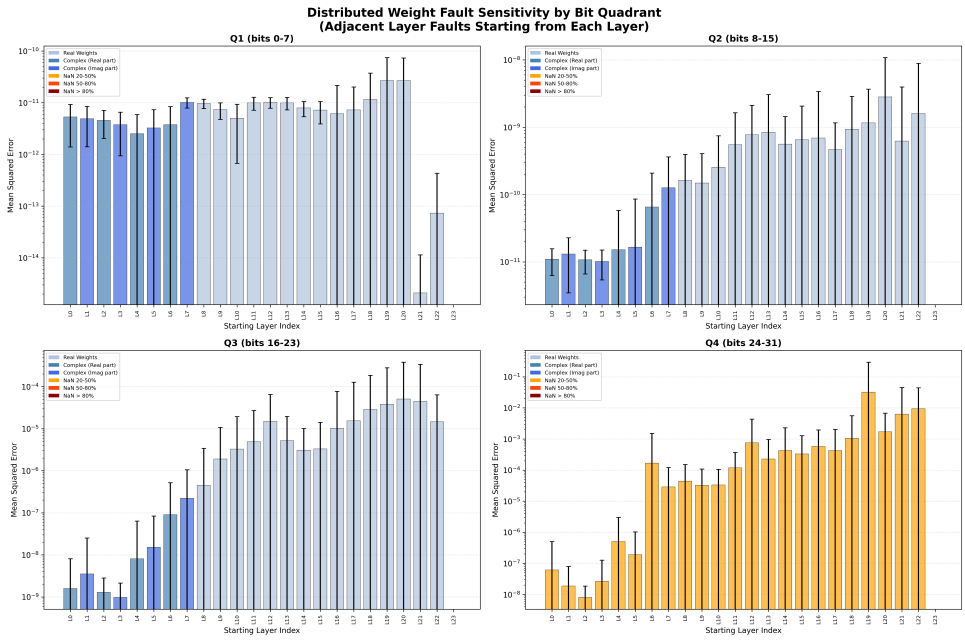

Figure 11: Distributed Weight Fault Sensitivity by bit-quadrants

Functional Interruptions (FIs)

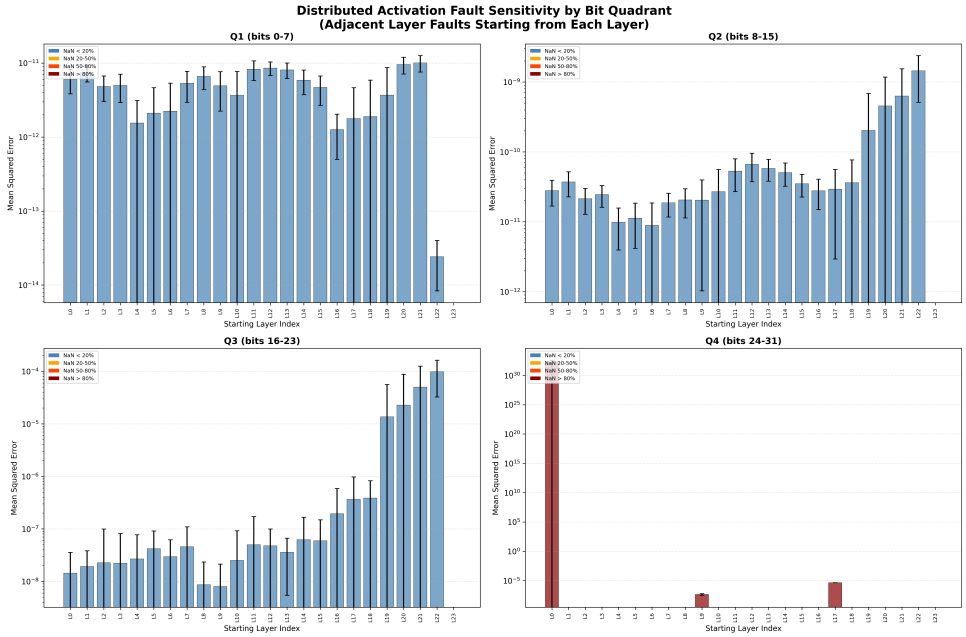

Figure 12: Distributed Activation Fault Sensitivity by bit-quadrants

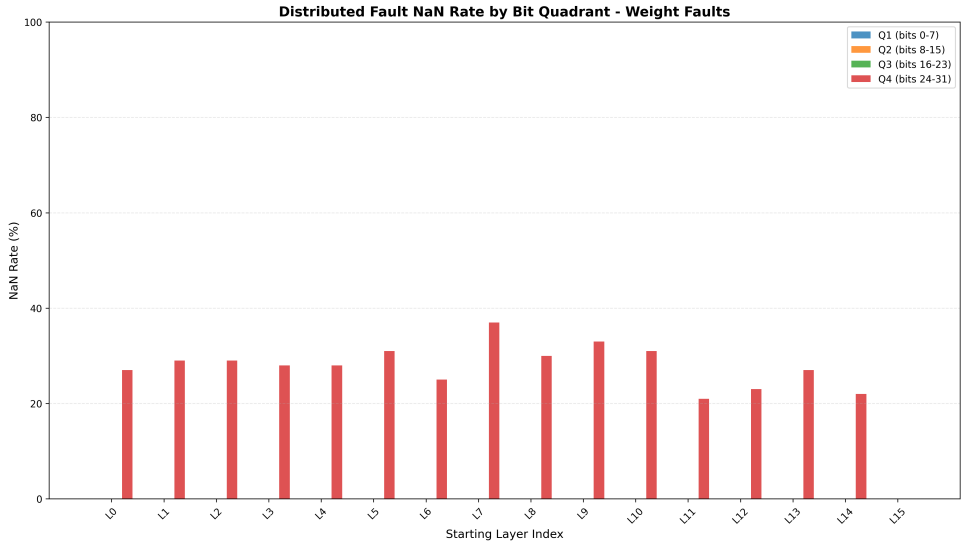

Figure 13: Functional Interruptions due to multi-bit fault injections in Weights in adjacent layers

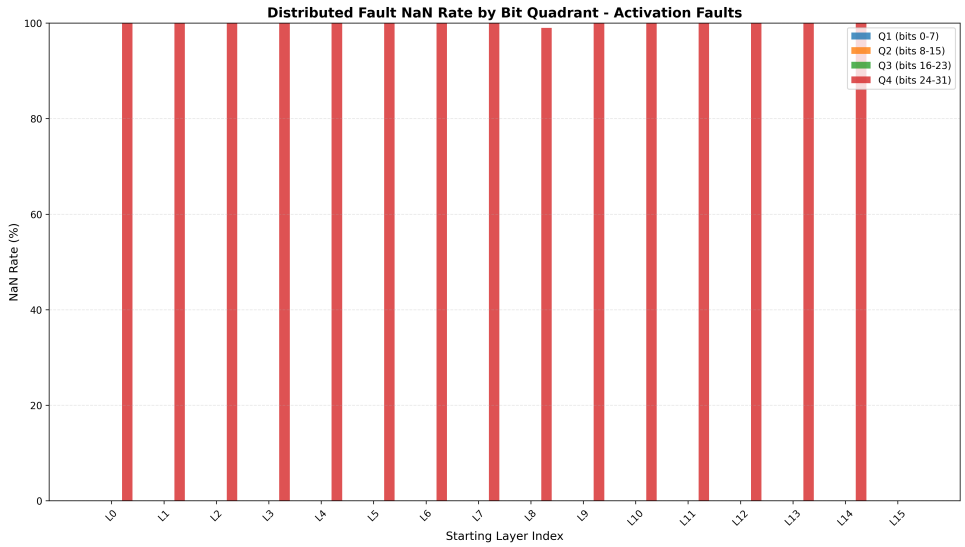

Figure 14: Functional Interruptions due to multi-bit fault injections in Activations in adjacent layers

3.3 Analysis of Results

The results of the simulation were collected as the delta MSE values between the original and fault injected model. The individual layer-wise information for weights and activations are captured in CSV files, and plots are generated to form an idea of the overall trend.

3.3.1 Silent Data Corruptions (SDCs)

The Silent Data Corruptions point to a changes in the average MSE difference of 10^{-11} to 10^{-4} from the original MSE value (baseline FNO) due to bit-flips in stored weights or activations, based on which bit-quadrant was getting corrupted.
Figure 1 shows the quadrant-wise distribution of MSE change, and Figure 2 captures the overall sensitivity of each layer to the weight upsets. Similarly, Figure 3 captures quadrant-wise information for activations, and Figure 4 captures the overall layer-wise sensitivity for activations.
Few interesting observations emerge here:

1. **Inherent Robustness of Spectral Convolution Layers**: The Spectral Convolution layers demonstrated significant resilience to bit-flip injections. The faults introduced in the spectralconv layers (columns 0-7: there are 4 spectralconv layers, each having real and complex weights) do not impact the MSE significantly. This observation is similar for baseline and Sloshing FNO. This robustness is an inherent property of the Fourier Neural Operator's architecture due to the global and distributed nature of the Fourier transform. The operation of SpectralConv layer involves:

   (a) Transforming the data to the frequency domain via FFT.

   (b) Multiplying by the (corrupted) weight.

   (c) Transforming back to the spatial domain via IFFT.

When an FFT is applied, the information from a single point in the spatial domain is distributed across all frequency components. A bit-flip in a single SpectralConv weight corrupts only one of these frequency components. When the IFFT is performed to return to the spatial domain, it is effectively an integration or summation over all frequency modes. The error from that single corrupted weight is averaged out and distributed globally across the entire output.
This finding strongly supports a **Selective protection strategy**, where TMR or other mitigation resources are not allocated to the SpectralConv layers, thereby achieving significant resource savings with minimal impact on overall reliability.

2. **High Vulnerability of Lifting and Projection Layers**: Lifting layers are the initial layers in the network that transform the dimensions of the input to pass it to the spectral layer. Projection layers are the final layers of the network. Any faults in these layers, especially in the Q4 bits (sign & exponent), seems to cause the most severe SDC. This observation is inline with studies on conventional neural networks, any faults in the input/output layer weights impact output significantly.

    - Lifting Layers: Being the first layer(s), any fault introduced in the lifting layers get amplified and propagated into the system and cause significant degradation at output.
    - Projection Layers: As the final layer(s), any fault introduced in projection layers do not get time to normalize, and get directly written to output.

3. **Low Functional Interrupt (FI) Rate from Weight Faults**: It is seen that most of the bit-flips in weights do not end up in Functional interruptions. This might be because when the weights are multiplied by inputs, they would even out out to a large, but still finite floating-point number that is still within FP limits.

The observations are similar for multi-bit fault injections, with only the ranges out delta MSE changing due to the bit flips.

3.3.2 Functional Interruptions (FI)

Functional Interruptions essentially halt the operation of the system, and for FNOs, a generation of NaN or inf values in the FP compute path can be considered as an FI. Using this context, the following trends are observed:

1. Faults in Activations, especially in the Q4 quadrant, have extremely high possibilities of resulting in an FI

2. The number of FIs is directly proportional to the number of faults introduced: Single bit flips are less prone to FI than multiple flips, which almost always result in FI, especially in Q4 bit quadrants.

Since Functional interruptions cause unnecessary stoppages to execution flow, a trade-off can be made whether to honor each FI generation, or overcome it with a small cost to accuracy and proceed with execution.

4 Improving SEU Fault Tolerance

The popular technique for improving SEU fault tolerance is redundancy, and Neural networks also adopt similar techniques, by deciding importance of parameters through various factors [11].
From the analysis of Fault Injection Campaigns on FNOs from the previous section, it is evident that the Spectral Convolution layers are inherently tolerant to atleast single bit upsets. To improve reliability without triplicating the entire system, strengthening layers other than SpectralConv should give acceptable fault-tolerance.
Towards this, an analysis was done on the FNO model (results are from the Sloshing model) to estimate the number of parameters present in non-Spectral Conv layers, and compute the estimated memory savings when strengthening only those parameters instead of triplicating all parameters. The details of the analysis are given below:

Parameter Breakdown

Component	**Parameters**
Total Model Parameters	18,887,843
Lifting	2,528
Projection	2,307
Skip Connections	4,224
Channel MLP	4,288
SpectralConv	18,874,496

TMR Protection

- Parameters Protected (non-spectral): 13,347
- Parameters Left Unprotected: 18,874,496 (SpectralConv)

TMR Implementation Overhead

- TMR Copies (3× replication): 40,041

Memory Savings: Full vs. Selective

Scenario	**Parameters**
Full Model TMR (3× all params)	56,663,529
Selective TMR (non-spectral only)	40,041
Selective vs Full	0.07%
Memory Savings	**99.93%**

Even considering the overhead due to majority voting logic, potential for memory savings is over 99%.

4.1 Selective Triple Modular Redundancy (TMR) Implementation

To mitigate the impact of soft errors on the FNO model, a selective Triple Modular Redundancy (TMR) scheme was implemented. This approach provides fault tolerance for the most vulnerable model components while minimizing the computational and memory overhead associated with full-model replication.

TMR is applied only to non-Spectral Convolution layers of the FNO, based on the understanding that SpectralConv layers exhibit inherent resilience:

- Lifting Layer (Encoder)
- Projection Layer (Decoder)
- Channel-wise MLPs (in each FNO block)
- Spatial Skip Connections (in each FNO block)

4.1.1 TMR Mechanism

The implementation uses 3 register buffers to store the same weight, and uses a Majority Voting Logic to identify the majority value of the 3 stored values.
This would ensure that a bit flip upset in any one bit will not affect the TMR output.

4.1.2 Verification and Impact on Inference Time

Post implementation, the functionality was verified via fault injections, and inference time benchmarks were run against the non-protected model to measure the performance overhead.

- **Functional Verification**
  1. A single-bit fault was injected into one of the three weight copies. The TMR model's output MSE remained identical to the fault-free baseline, demonstrating successful correction.
  2. Identical bit-flips were injected into two of the three weight copies. TMR will be unable to correct errors in multiple copies of the same bit.
- **Performance Overhead** Using register buffers and a majority voting logic is bound to negatively impact the inference time. The wall-clock time of the TMR-protected model was compared to the unprotected model over $N = 100$ samples, quantifying the latency overhead of the read-vote-write scrubbing step performed each forward pass.

Benchmark Results for TMR-Protected vs Original Model

```
-------------------------
BENCHMARK RESULTS
--------------------------------------------------------------------------------
Original Model:
```

```
  Total time:       0.2690 seconds
  Time per sample:  2.69 ms
  Throughput:       371.79 samples/sec

TMR Model:
  Total time:       0.3853 seconds
  Time per sample:  3.85 ms
  Throughput:       259.51 samples/sec

Overhead:
  Additional time:  0.1164 seconds
  Time overhead:    43.27%
  Slowdown factor:  1.43x
--------------------------------------------------------------------------------
```

There is degradation on performance caused by the TMR inference, but the slowdown is acceptable considering the increased reliability TMR offers.

4.2 Activation Clipping

From the analysis of Activation Fault injection, it is clear that perturbations to activations can result in Functional Interruptions more frequently than faults in weights. This would halt the entire system, which may not be ideal. Some systems may prefer to continue operation, even if at the cost of reduced accuracy.

In this line of thought, we can consider a function that prevents activations from triggering Functional Interruptions. Activation clipping has been used in standard DNNs to improve reliability [12], in this case, we can utilise it to prevent Functional interruptions and ensure that the system does not halt.

For systems with a limited input range, like the scientific problems of Darcy or Sloshing, we can assume the inputs and hence the max values of activations to be bounded within a certain range. We estimate the max Activation values layerwise and implement a clipping function, that will limit the maximum activation value during inference to this selected value. To strengthen the system further, the storage for activation clipping values can be TMR-ed (not done in this implementation).

The system was tested with fault injection in all layers, and was found to remove all Functional Interruptions (due to NaN or inf generation), at an average MSE loss of 1.56 x 10^{-4} (for Baseline).

```
================================================================================
                    SUMMARY: NaN REDUCTION AND MSE IMPACT
================================================================================
Quadrant          NaN%       NaN%  Saved%  MSE (no clip)        MSE (clip)
--------------------------------------------------------------------------------
Q1 (bits 0-7)     0.0%   →   0.0%   0.0%   2.838200e-12        2.838200e-12
Q2 (bits 8-15)    0.0%   →   0.0%   0.0%   8.847300e-11        8.847300e-11
Q3 (bits 16-23)   0.0%   →   0.0%   0.0%   5.718615e-06        4.644065e-06
Q4 (bits 24-31)  95.3%   →   0.0%  95.3%   2.256759e+30        1.560731e-04
================================================================================
```

5 Implementation on Raspberry Pi

As part of edge implementation for FNO, the baseline FNO trained on Darcy flow was implemented on a Raspberry Pi v3 with a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB of RAM.
The Raspberry Pi was setup with the following version of Linux (Debian):

```
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
NAME="Debian GNU/Linux"
VERSION_ID="12"
VERSION="12 (bookworm)"
VERSION_CODENAME=bookworm
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

The following packages were necessary for running FNO:

```
Package            Version
------------------ -----------
annotated-types    0.7.0
certifi            2025.10.5
charset-normalizer 3.4.4
click              8.3.0
configmypy         0.2.0
contourpy          1.3.3
cycler             0.12.1
filelock           3.20.0
fonttools          4.60.0
fsspec             2025.9.0
gitdb              4.0.12
GitPython          3.1.45
h5py               3.14.0
idna               3.11
iniconfig          2.3.0
```

Jinja2	3.1.6
kiwisolver	1.4.9
MarkupSafe	3.0.3
matplotlib	3.10.6
mpmath	1.3.0
networkx	3.5
neuraloperator	2.0.0
neuraloperators	1.0
numpy	2.3.3
opt_einsum	3.4.0
packaging	25.0
pandas	2.3.3
pillow	11.3.0
pip	25.2
platformdirs	4.5.0
pluggy	1.6.0
protobuf	6.33.0
pydantic	2.12.3
pydantic_core	2.41.4
Pygments	2.19.2
pyparsing	3.2.5
pytest	8.4.2
pytest-mock	3.15.1
python-dateutil	2.9.0.post0
pytorchfi	0.6.0
pytz	2025.2
PyYAML	6.0.3
requests	2.32.5
ruamel.yaml	0.18.16
ruamel.yaml.clib	0.2.14
scipy	1.16.2
sentry-sdk	2.42.1
setuptools	66.1.1
six	1.17.0
smmap	5.0.2
sympy	1.14.0
tensorly	0.9.0
tensorly-torch	0.5.0
torch	2.9.0
torchaudio	2.9.0
torchvision	0.24.0
tqdm	4.67.1
typing_extensions	4.15.0
typing-inspection	0.4.2
tzdata	2025.2
urllib3	2.5.0

```
wandb               0.22.2
zencfg              0.6.0
```

Once the OS and packages were installed, the baseline FNO was run on the Raspberry Pi. It completed execution, although at a much lower performance of about 2.8 frames per second, as opposed to the 300+ FPS from the server machine. As seen from the logs, the performance of FNO on edge is significantly lesser, due to the heavy compute involved. Future directions could consider improving the performance of FNO on RaspPi.
The TMR version of the FNO was also ported and run on the RPi, to a surprising result:

```
--------------------------------------------------------------------------------
BENCHMARK RESULTS
--------------------------------------------------------------------------------
Original Model:
  Total time:       33.9986 seconds
  Time per sample:  339.99 ms
  Throughput:       2.94 samples/sec

TMR Model:
  Total time:       36.1497 seconds
  Time per sample:  361.50 ms
  Throughput:       2.77 samples/sec

Overhead:
  Additional time:  2.1511 seconds
  Time overhead:    6.33%
  Slowdown factor:  1.06x
--------------------------------------------------------------------------------
```

The slowdown in RPi with TMR was only 1.06x compared to 1.43x in the Server! This was later understood to be a manifestation of the (in)famous Amdahl's Law:

- There are two main operations during TMR execution: The compute-bound inference part, and the memory-bound TMR copy access part.
- Raspberry Pi takes 30+ seconds for the compute part, while the Server takes just around 0.3 seconds.
- Raspberry Pi and Server take similar time for memory access (Additional access times are 0.17ns and 0.12ns)

Therefore, compared to the total time, the overhead in the RPi is much less than that in the Server, since both are limited by the non-enhancable memory access, even though the Server is much faster in compute.

Figure 15: Raspberry Pi running Fourier Neural Operators

6 Future

The Fault Injection campaigns have provided an idea of the resilience of different layers of the Fourier Neural Network. Future prospects can focus in these directions:

- **Code-book based Resilience**: If there is potential in the network to quantize the Spectral convolution layers, a standard code-book based implementation for the complex weights can be considered. Triplicating these code-book values would incur significantly less penalty than system-level TMR.
- **Pruning for FNOs**: Despite the perturbation in Spectral convolution weights not significantly affecting the output, there is limited research on pruning Fourier Neural Operators. This is an area worth exploring, at least to understand fundamental limitations that gate pruning.
- **Improving Edge Performance**: Given that FNOs still take a lot of time on edge devices (though better than conventional solvers), the complex float32 weight based computation takes a toll. It can be seen if quantization to int values would aid inference time in such devices with limited loss to accuracy.

A Appendix

A.1 Fault Injection Campaign Results for Sloshing FNO

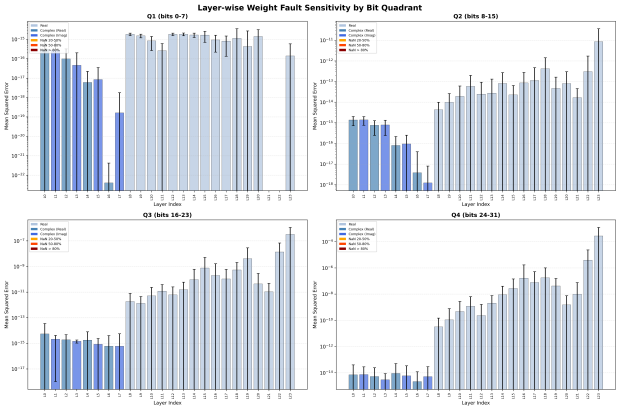

Figure 16: Layer-wise sensitivity to single-bit upset in Weights (Slosh FNO)

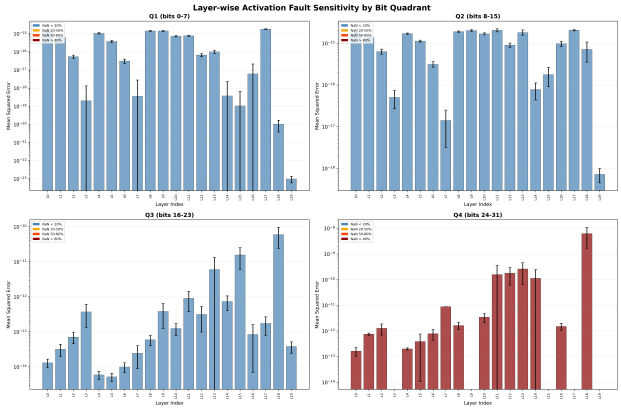

Figure 17: Layer-wise sensitivity to single-bit upset in Activations (Slosh FNO)

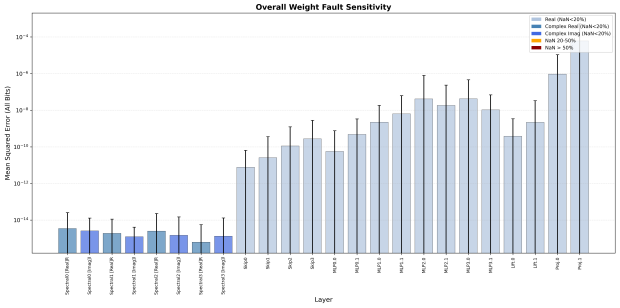

Figure 18: Overall Layer-wise sensitivity to single-bit upset in Weights (Slosh FNO)

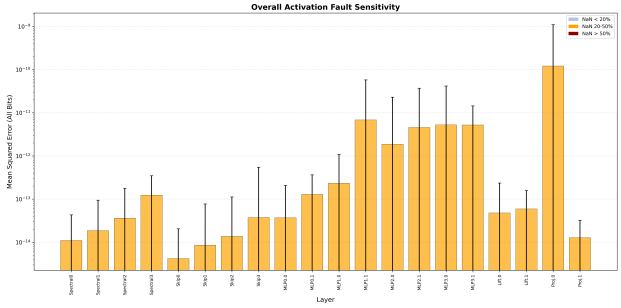

Figure 19: Overall Layer-wise sensitivity to single-bit upset in Activations (Slosh FNO)

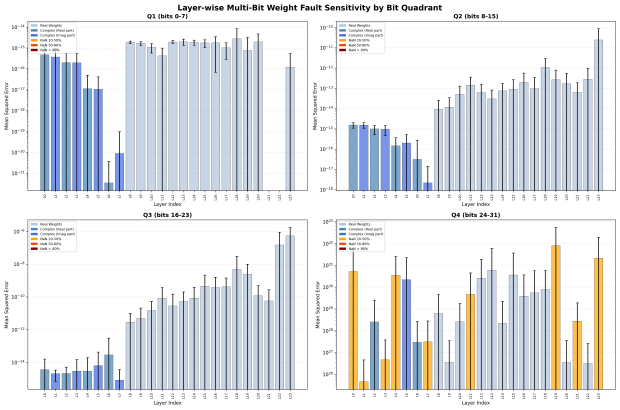

Figure 20: Layer-wise Sensitivity to Concentrated MBU in Weights (Slosh FNO)

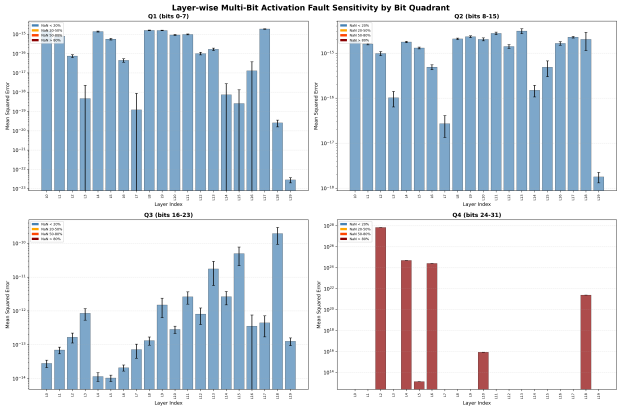

Figure 21: Layer-wise Sensitivity to Concentrated MBU in Activations (Slosh FNO)

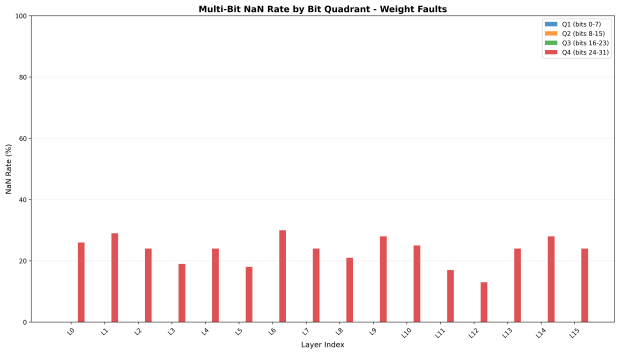

Figure 22: FI due to Concentrated MBU in Weights (Slosh FNO)

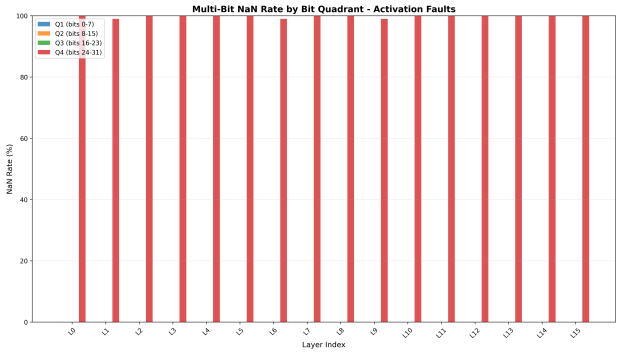

Figure 23: FI due to Concentrated MBU in Activations (Slosh FNO)

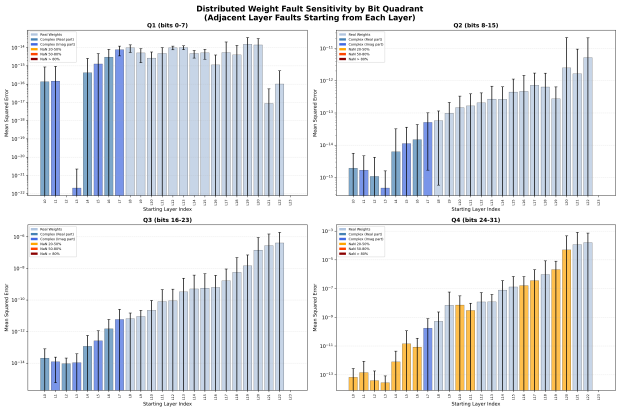

Figure 24: Layer-wise Sensitivity to Distributed MBU in Weights (Slosh FNO)

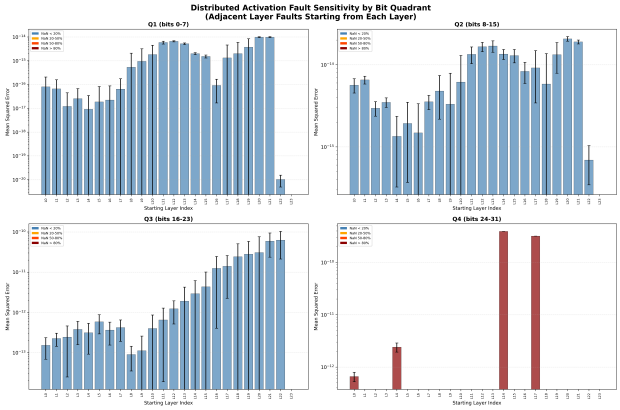

Figure 25: Layer-wise Sensitivity to Distributed MBU in Activations (Slosh FNO)

A.2 GitHub Link to Project

https://github.com/rahulak-cs-iitm/cs6886-jul-nov-2025-course-project-rephino/

References

[1] T. Bonnoit, A. Coelho, N.-E. Zergainoh, and R. Velazco, "Seu impact in processor's control-unit: Preliminary results obtained for leon3 soft-core," in *2017 18th IEEE Latin American Test Symposium (LATS)*, pp. 1–4, 2017.

[2] M. Barbirotta, F. Menichelli, A. Cheikh, A. Mastrandrea, M. Angioli, and M. Olivieri, "Dynamic triple modular redundancy in interleaved hardware threads: An alternative solution to lockstep multi-cores for fault-tolerant systems," *IEEE Access*, vol. 12, pp. 95720–95735, 2024.

[3] E. Baviera, G. M. Schettino, E. Tuniz, and F. Vatta, "Software implementation of error detection and correction against single-event upsets," in *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–6, 2020.

[4] M. S. Gorbunov, "Design of fault-tolerant microprocessors for space applications," *Acta Astronautica*, vol. 163, pp. 252–258, Oct. 2019.

[5] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[6] W. Mansour and R. Velazco, "Seu fault-injection in vhdl-based processors: A case study," in *2012 13th Latin American Test Workshop (LATW)*, pp. 1–5, 2012.

[7] G. S. Rodrigues, F. Rosa, B. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the impact of fault-tolerance methods in arm processors under soft errors running linux and parallelization apis," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2196–2203, 2017.

[8] Z. Yan, Y. Shi, W. Liao, M. Hashimoto, X. Zhou, and C. Zhuo, "When single event upset meets deep neural networks: Observations, explorations, and remedies," 2019.

[9] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 25–31, 2020.

[10] G. Swift and S. Guertin, "In-flight observations of multiple-bit upset in drams," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2386–2391, 2000.

[11] K. Soroush, N. Shirazi, and M. Raji, "Efficient triple modular redundancy for reliability enhancement of dnns using explainable ai," 07 2025.

[12] L.-H. Hoang, M. A. Hanif, and M. Shafique, "Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1241–1246, 2020.