SliceFI: A Heuristic-Based Fault Injection Tool for Edge Neural Networks

Andy Meza, Olivia Weng, Nhan Tran, Ryan Kastner

Presented at A3D3 All-Hands Annual Meeting on August 16, 2025 in Pasadena, California

UC San Diego

Fermilab

Why analyze the **sensitivity** of edge NNs?

Why analyze the sensitivity of edge NNs?

- Edge NNs can be sensitive creatures

Why analyze the sensitivity of edge NNs?

- Edge NNs can be sensitive creatures
- They are put through a lot:

Why analyze the sensitivity of edge NNs?

- Edge NNs can be sensitive creatures

- They are put through a lot:
    - Pruning
    - Quantization
    - Hardware faults

Why analyze the sensitivity of edge NNs?

- Edge NNs can be sensitive creatures

- They are put through a lot:
  - Pruning
  - Quantization
  - **Hardware faults**

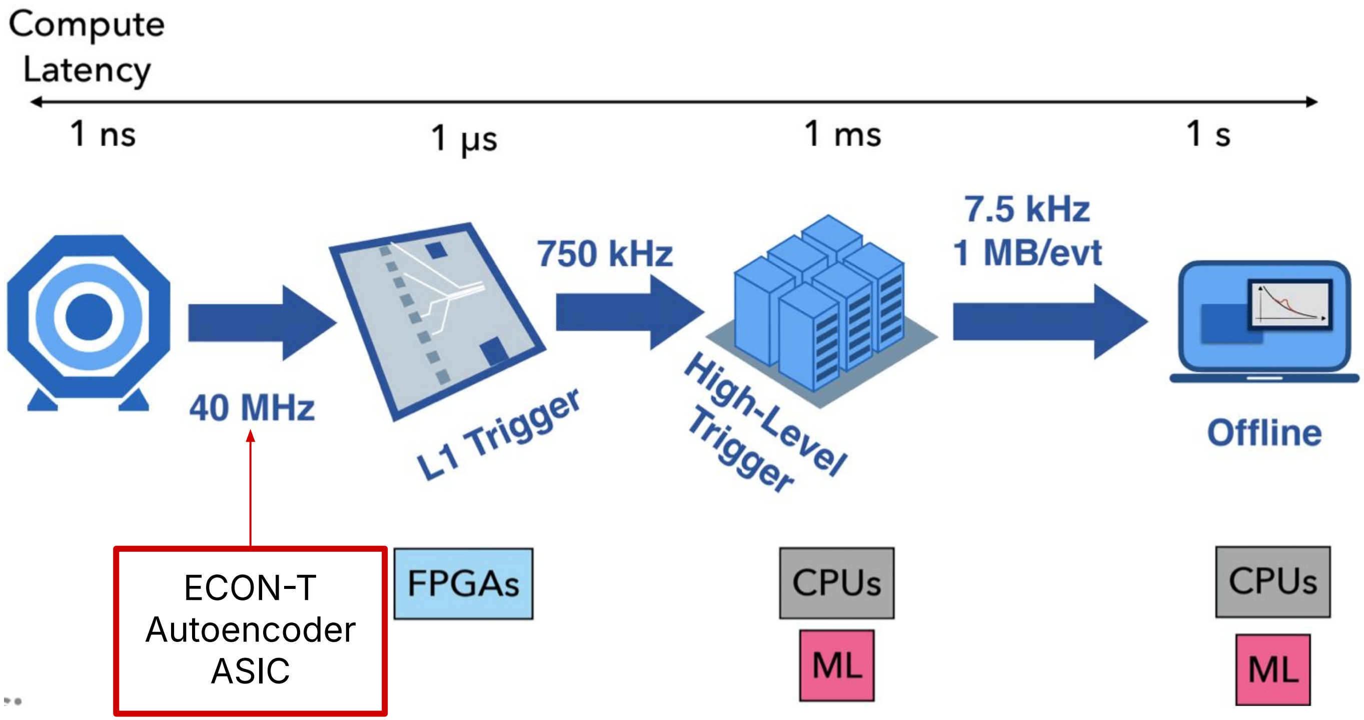
Why analyze the sensitivity of edge NNs?

- Edge NNs can be sensitive creatures

- They are put through a lot:
  - Pruning
  - Quantization
  - **Hardware faults**

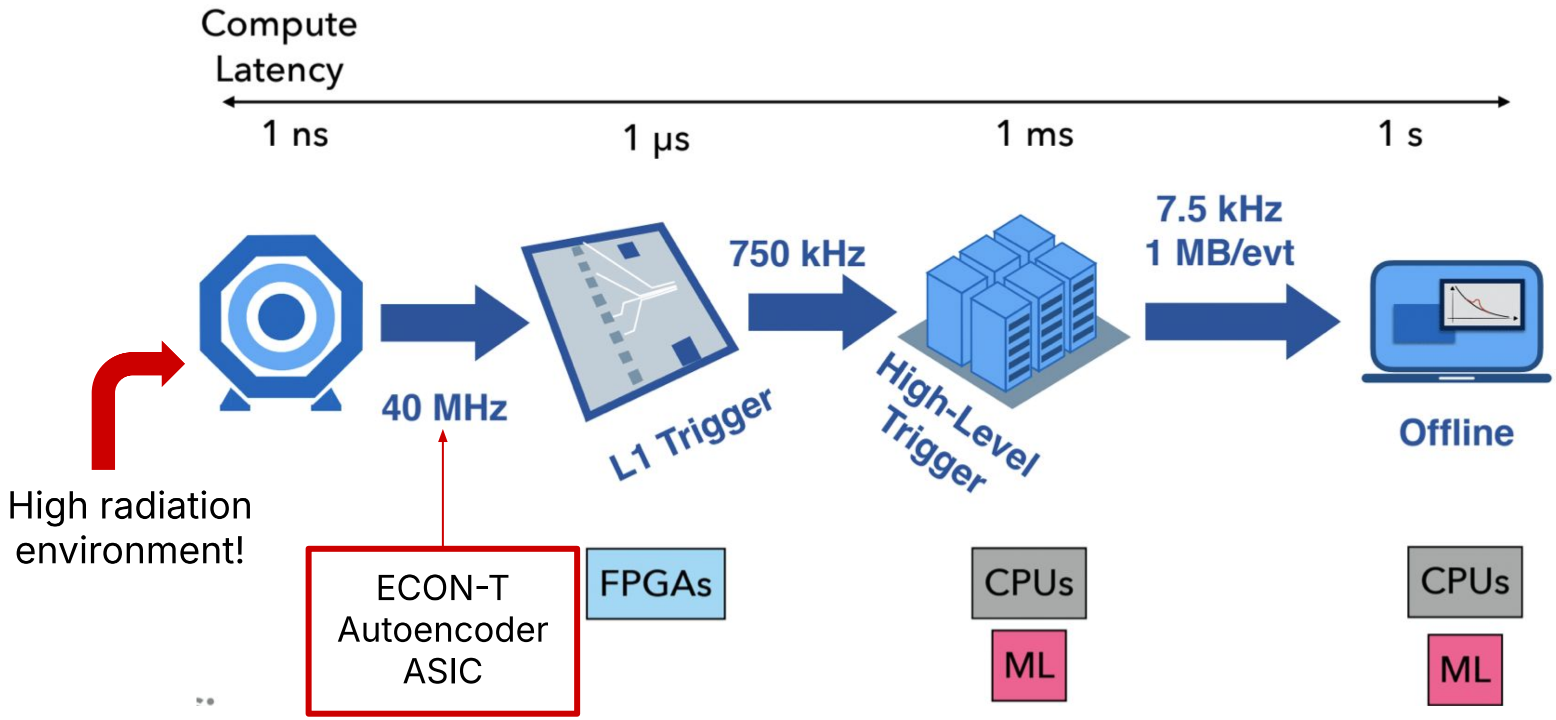
When do **hardware faults** occur?

Example: LHC's CMS Data Processing Pipeline



Ref: https://indico.fnal.gov/event/46746/contributions/210450/attachments/141293/177902/hirschauer_AE_CPAD_19mar2020.pdf

Example: LHC's CMS Data Processing Pipeline



Ref: https://indico.fnal.gov/event/46746/contributions/210450/attachments/141293/177902/hirschauer_AE_CPAD_19mar2020.pdf

How does the ECON-T Autoencoder **tolerate** radiation?

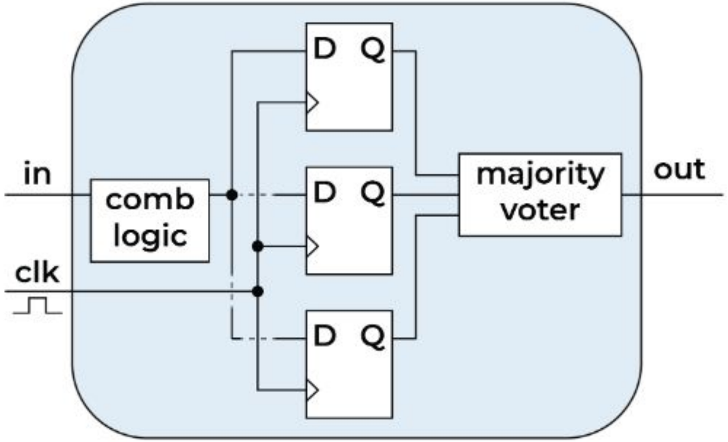
ECON-T Radiation Tolerance

- ECON-T employs **triple modular redundancy (TMR)** to its registers

Ref: Di Guglielmo et al. A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. IEEE Trans. Nucl. Sci.'21.

ECON-T Radiation Tolerance

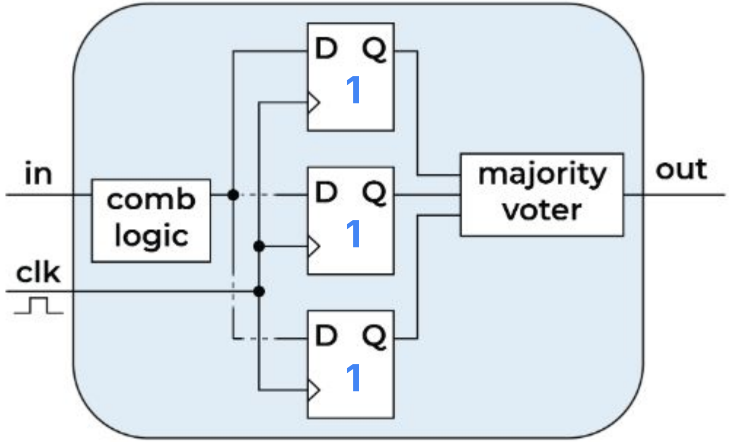
- ECON-T employs **triple modular redundancy (TMR)** to its registers



Ref: Di Guglielmo et al. A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. IEEE Trans. Nucl. Sci.'21.

ECON-T Radiation Tolerance

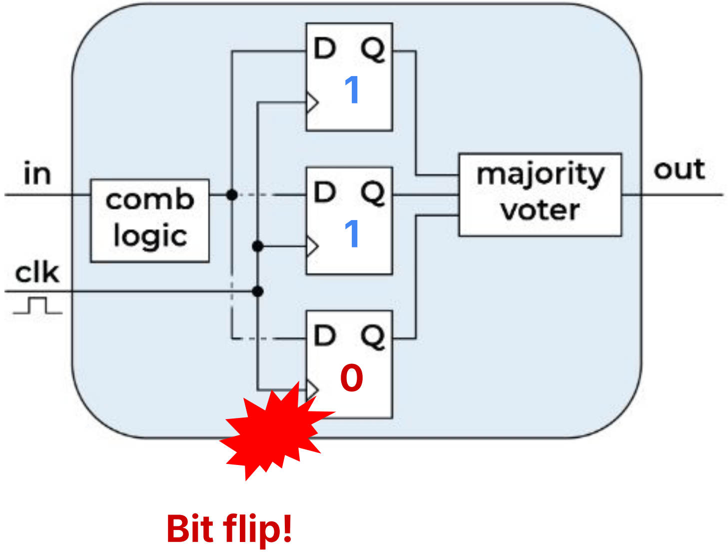
- ECON-T employs **triple modular redundancy (TMR)** to its registers



Ref: Di Guglielmo et al. A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. IEEE Trans. Nucl. Sci.'21.

ECON-T Radiation Tolerance

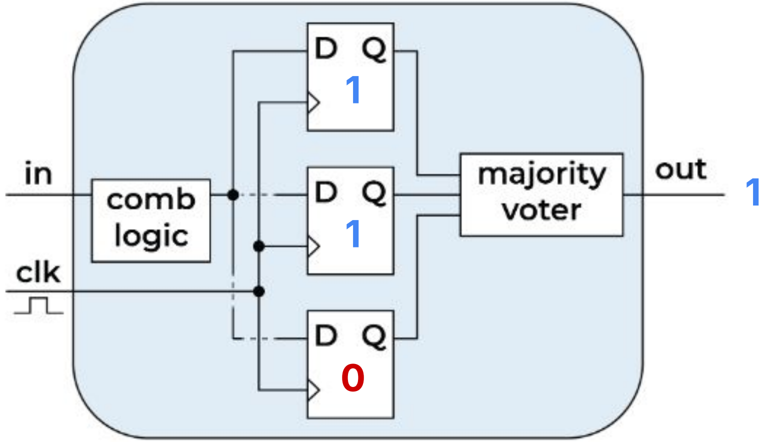
- ECON-T employs **triple modular redundancy (TMR)** to its registers



Ref: Di Guglielmo et al. A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. IEEE Trans. Nucl. Sci.'21.

ECON-T Radiation Tolerance

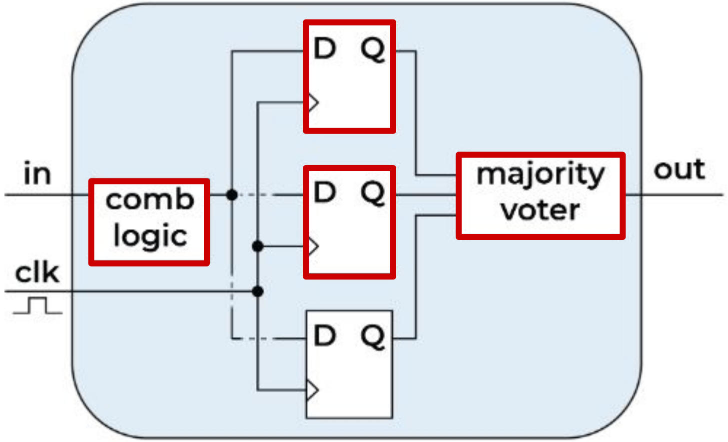
- ECON-T employs **triple modular redundancy (TMR)** to its registers



Ref: Di Guglielmo et al. A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. IEEE Trans. Nucl. Sci.'21.

ECON-T Radiation Tolerance

- ECON-T employs **triple modular redundancy (TMR)** to its registers



TMR incurs ≥200% area overhead!

Ref: Di Guglielmo et al. A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. IEEE Trans. Nucl. Sci.'21.

How can we **reduce** radiation tolerance **costs**?

Observation: Tolerance only applied to **hardware**

Observation: Tolerance only applied to **hardware**

What about **software**?

How should we assess
the **fault sensitivity** of NN **software**?

How should we assess
the **fault sensitivity** of NN **software**?

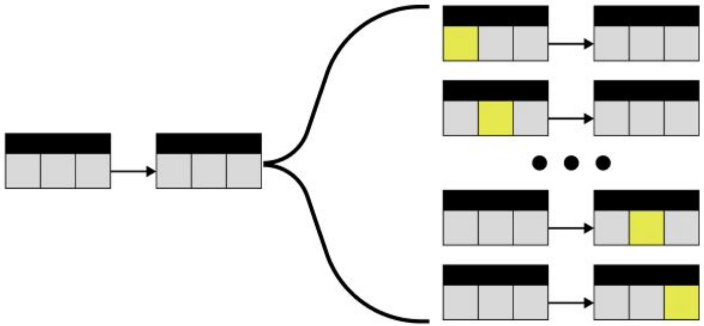
Fault Injection Campaigns

Fault Injection Campaigns

- Conceptually, each fault campaign:

Fault Injection Campaigns

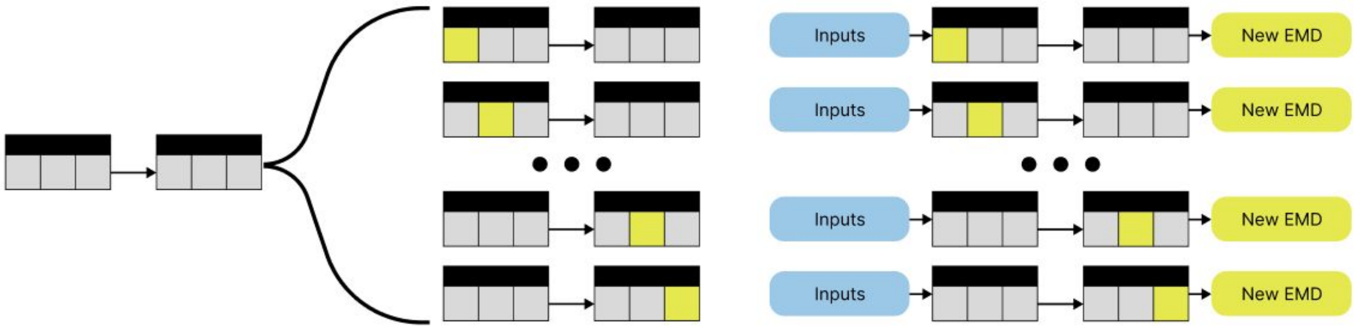
- Conceptually, each fault campaign:



1. Generates X "faulty" variants by flipping a single weight bit

Fault Injection Campaigns

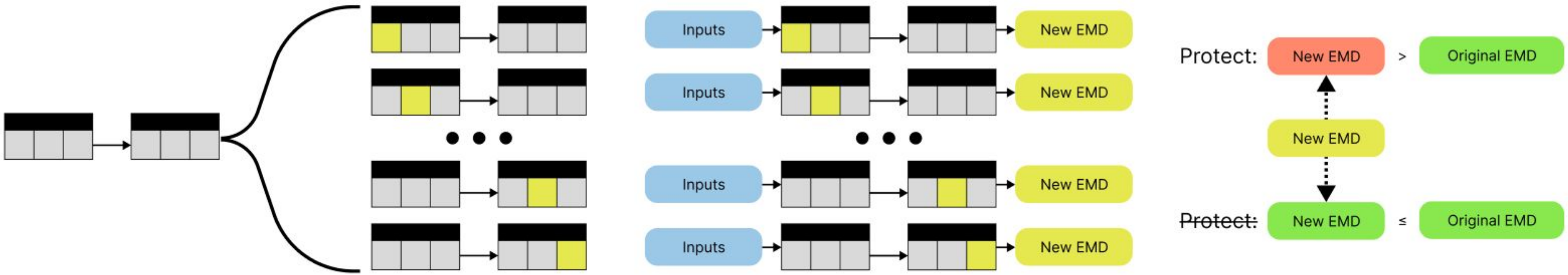
- Conceptually, each fault campaign:



1. Generates X "faulty" variants by flipping a single weight bit 2. Measures the new EMD on a set of test inputs

Fault Injection Campaigns

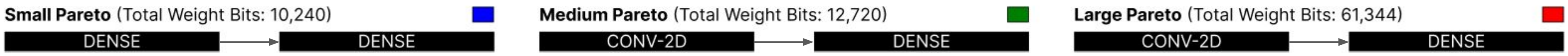
- Conceptually, each fault campaign:



1. Generates X “faulty” variants by flipping a single weight bit
2. Measures the new EMD on a set of test inputs
3. Determines the weight bits to protect

Fault Injection Campaign on ECON-T

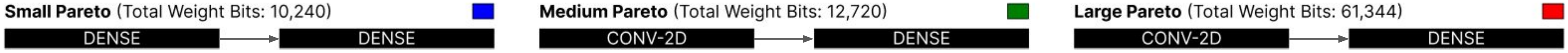
- Previously, we've conducted fault injection campaigns on 3 ECON-T variants

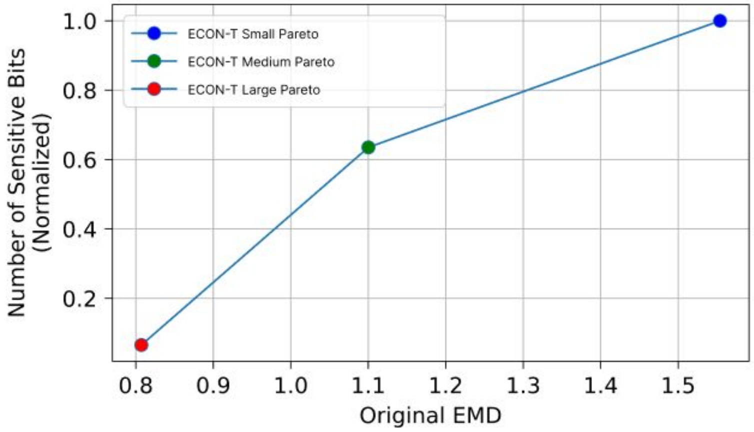


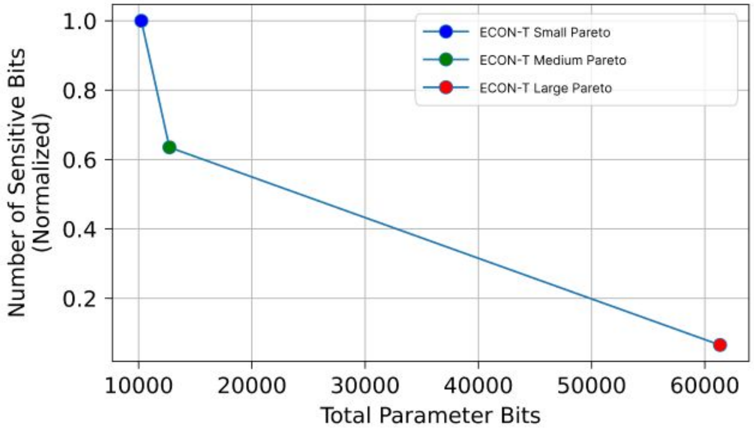
Ref: Weng, Olivia, Meza, Andres, et al. "Fkeras: A sensitivity analysis tool for edge neural networks." JATS 1.3 (2024): 1-27

Fault Injection Campaign on ECON-T

- Previously, we've conducted fault injection campaigns on 3 ECON-T variants







Ref: Weng, Olivia, Meza, Andres, et al. "Fkeras: A sensitivity analysis tool for edge neural networks." JATS 1.3 (2024): 1-27

Fault injection campaigns are expensive

Fault injection campaigns are expensive

Why?

Fault Injection Campaigns are Expensive...

- Because:
  - Must flip every bit in the model
  - For every bit flip:
    - Full inference is run on every sample of a validation set
- High-level Analysis of FI Campaign

```
results = list()
for bi in model_bits:
    faulty_model = model.flip_bit(bi)
    for batch in validation set:
        faulty_output = faulty_model.predict(batch)
        result.append(faulty_output)
. . .
```

Fault Injection Campaigns are Expensive...

- Because:
    - Must flip every bit in the model
    - For every bit flip:
        - Full inference is run on every sample of a validation set
- High-level Analysis of FI Campaign

```
results = list()
for bi in model_bits:
    faulty_model = model.flip_bit(bi)
    for batch in validation set:
        faulty_output = faulty_model.predict(batch) ← Most Expensive Operation!
        result.append(faulty_output)
. . .
```

Fault Injection Campaigns are Expensive...

- High-level Analysis of FI Campaign

```
results = list()
for bi in model_bits:
    faulty_model = model.flip_bit(bi)
    for batch in validation set:
        faulty_output = faulty_model.predict(batch) ← Most Expensive Operation!
        result.append(faulty_output)
. . .
```

- A closer look at inference

$$X_4 = L_4(L_3(L_2(L_1(X_0))))$$

$$X_0 \rightarrow L_1 \rightarrow X_1 \rightarrow L_2 \rightarrow X_2 \rightarrow L_3 \rightarrow X_3 \rightarrow L_4 \rightarrow X_4$$

Fault Injection Campaigns are Expensive...

- High-level Analysis of FI Campaign

```
results = list()
for bi in model_bits:
    faulty_model = model.flip_bit(bi)
    for batch in validation set:
        faulty_output = faulty_model.predict(batch) ← Most Expensive Operation!
        result.append(faulty_output)
. . .
```

- A closer look at inference

$$X_4 = L_4(L_3(L_2(L_1(X_0))))$$

$$X_0 \rightarrow L_1 \rightarrow X_1 \rightarrow L_2 \rightarrow X_2 \rightarrow L_3 \rightarrow X_3 \rightarrow L_4 \rightarrow X_4$$

Do we need to run full inference for every bit flip?

Is Full Inference Necessary?

- Maybe not!
- Key Observations:
  - Each bit flip only changes one layer
  - But we execute all layers in the model regardless of which bit has been flipped

Is Full Inference Necessary?

- Maybe not!
- Key Observations:
  - Each bit flip only changes one layer
  - But we execute all layers in the model regardless of which bit has been flipped
- Idea:
  - Could we infer the behavior at the model output from analyzing only a single layer's output?

$$X_4 = L_4(L_3(L_2(L_1(X_0))))$$

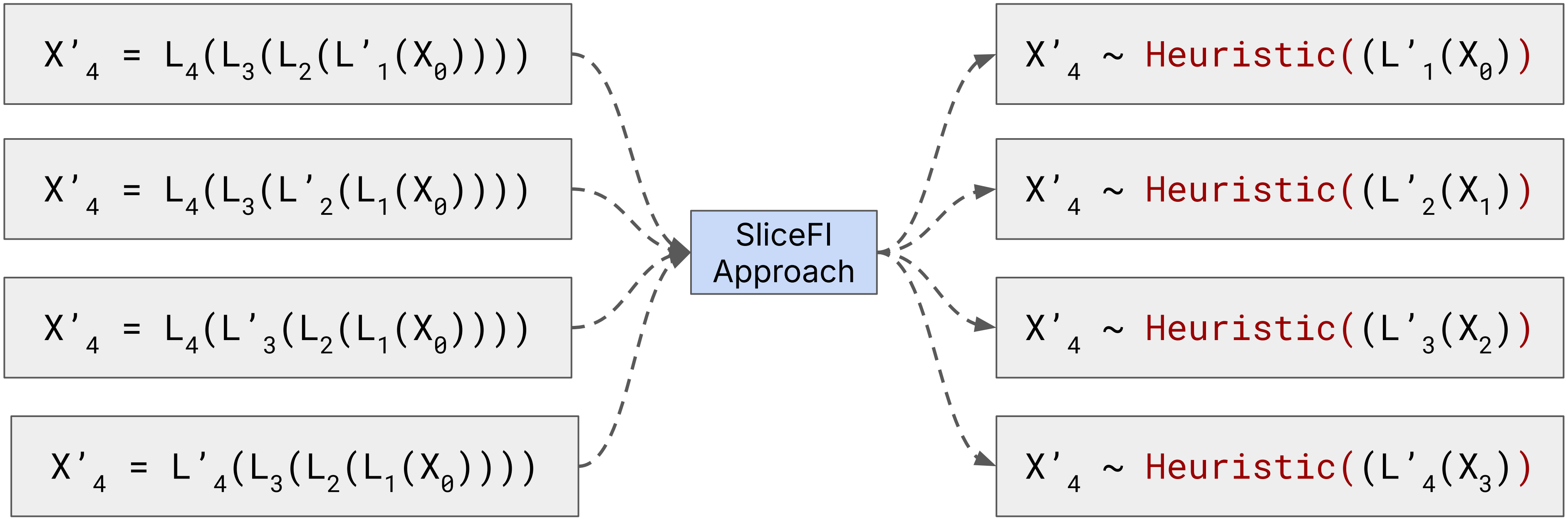
$$X_0 \rightarrow L_1 \rightarrow X_1 \rightarrow L_2 \rightarrow X_2 \rightarrow L_3 \rightarrow X_3 \rightarrow L_4 \rightarrow X_4$$

$$X_4 = \mathtt{Heuristic}((L_1(X_0)))$$

$$X_0 \rightarrow L_1 \rightarrow X_1 \rightarrow \mathtt{Heuristic} \rightarrow {\sim}X_4$$

SliceFI

- An FI tool that approximates full faulty inference from a faulty model layer
- SliceFI slices a model into individual layers (slices) to speed up FI campaigns



SliceFI Procedure

- Slice → Cache Expected Results → Run FI Campaign with Heuristic

```
expect_slice_outputs, slices = slice_fi(model)
results = list()
for bi in model_bits:
    slices[bi].flip_bit(bi)
    for batch in validation set:
        slice_output = slices[bi].predict(batch)
        faulty_output = heuristic(slice_output, expected_slice_outputs[bi])
        result.append(faulty_output)
. . .
```

SliceFI Procedure

- Slice → Cache Expected Results → Run FI Campaign with Heuristic

```
expect_slice_outputs, slices = slice_fi(model)
results = list()
for bi in model_bits:
    slices[bi].flip_bit(bi)
    for batch in validation set:
        slice_output = slices[bi].predict(batch) ← Cheaper Operation!
        faulty_output = heuristic(slice_output, expected_slice_outputs[bi])
        result.append(faulty_output)
. . .
```

- Key Differences:
    - Inference is performed on a smaller “model” (i.e., a layer)
    - We do not recompute layers upstream or downstream of where fault is being injected

What's a Good Heuristic?

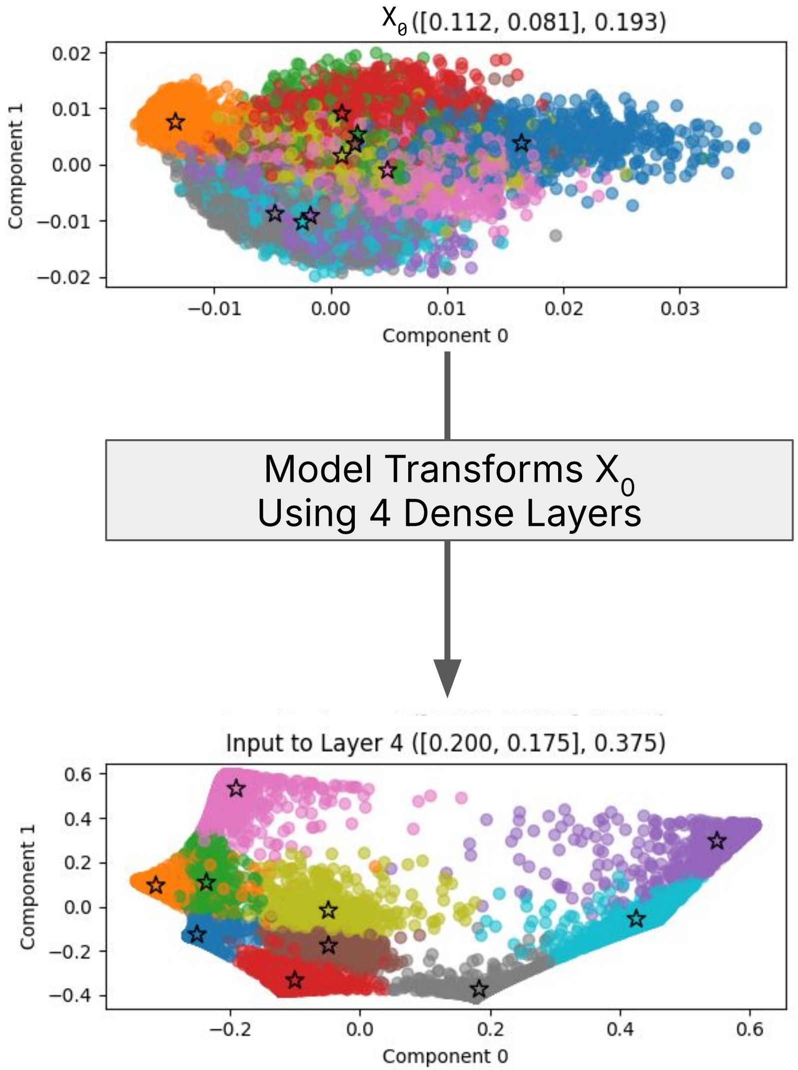
- Goal:
  - Avoid recomputing layers upstream of where fault is being injected
    - Easily solved with caching
  - Avoid recomputing layers downstream of where fault is being injected
    - Challenge
    - Heuristic needs to enable us to infer behavior at model output from analyzing layer output

What’s a Good Heuristic?

- Goal:
  - Avoid recomputing layers upstream of where fault is being injected
    - Easily solved with caching
  - Avoid recomputing layers downstream of where fault is being injected
    - Challenge
    - Heuristic needs to enable us to infer behavior at model output from analyzing layer output
- Idea:
  - We have a set of expected coordinates for each validation sample as it passes through the model
  - A fault injection may alter the expected coordinates (i.e., faulty coordinates)
  - A distance/difference-based heuristic on the expected vs faulty layer outputs may capture behavior at the model output

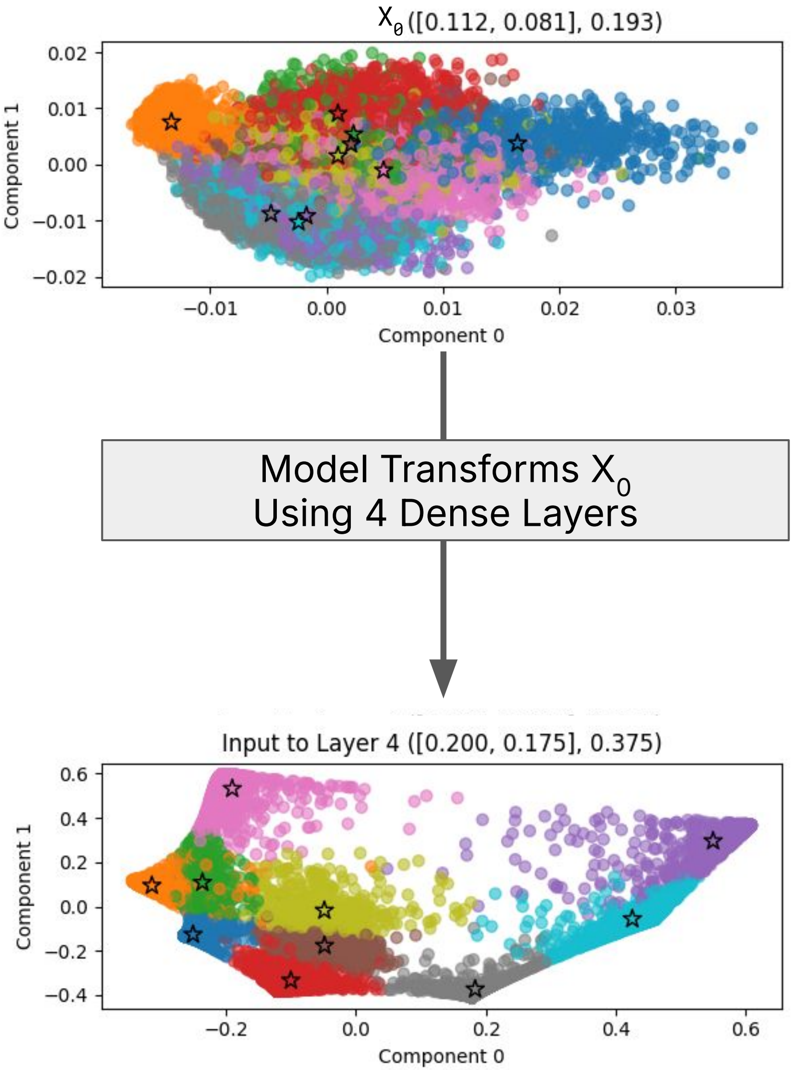
MNIST Model

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$



MNIST Model

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$
- Scatter Plot Details:
  - PCA reduction used to make 2D scatter plot
  - Points are colored by GT label/digit
  - Stars denote the centroid of GT clusters

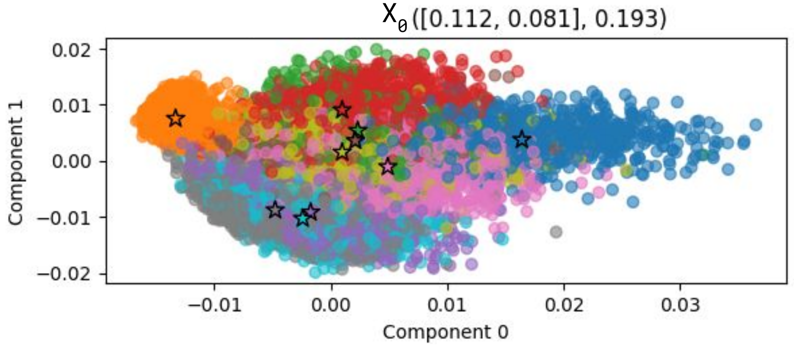


MNIST Model Expected Results

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$

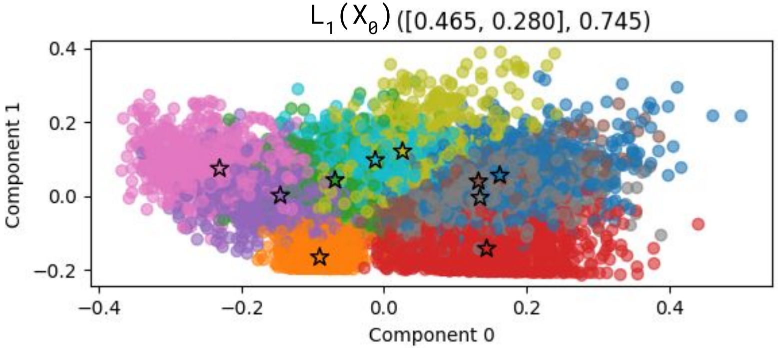
MNIST Model Expected Results

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$



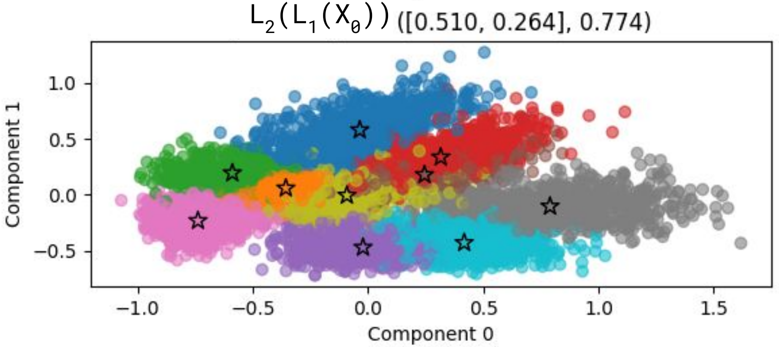
MNIST Model Expected Results

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$



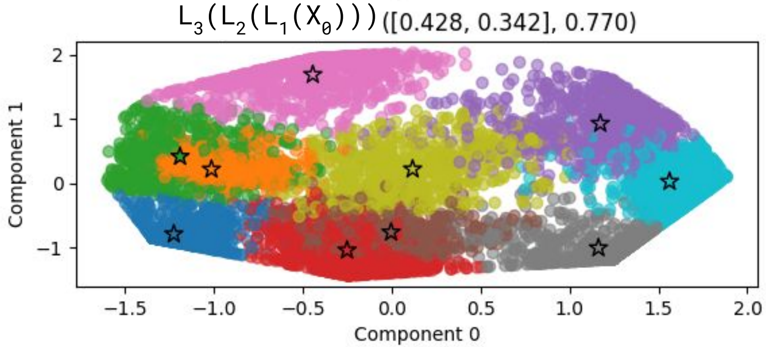
MNIST Model Expected Results

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$



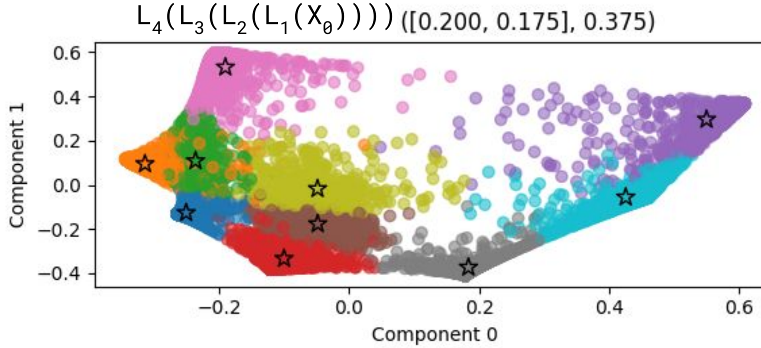
MNIST Model Expected Results

- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$



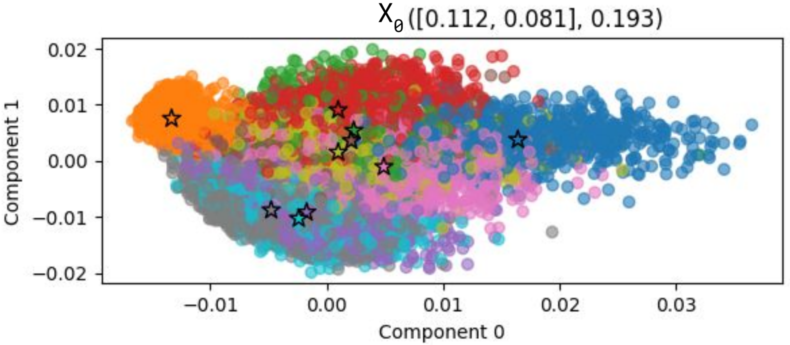
MNIST Model Expected Results

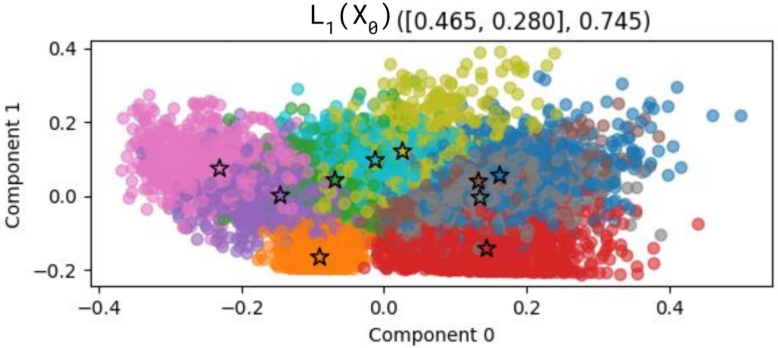
- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$

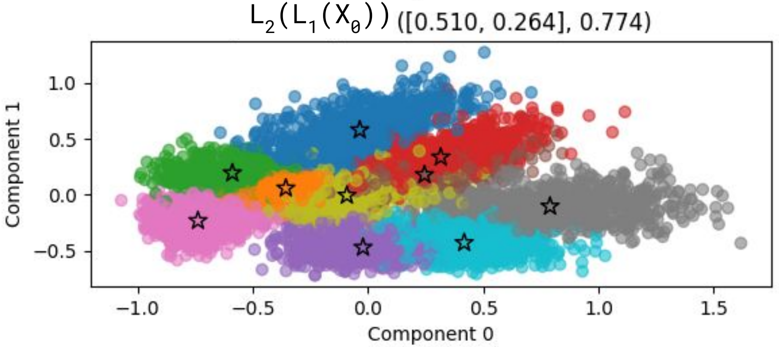


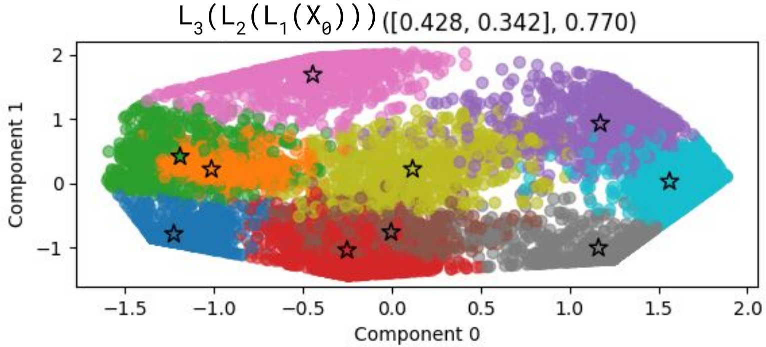
MNIST Model Expected Results

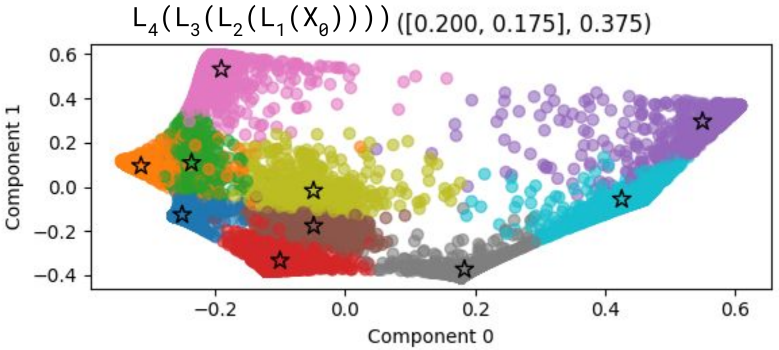
- MNIST Task: Classify X_0 as a digit 0-9
- $X_4 = L_4(L_3(L_2(L_1(X_0))))$









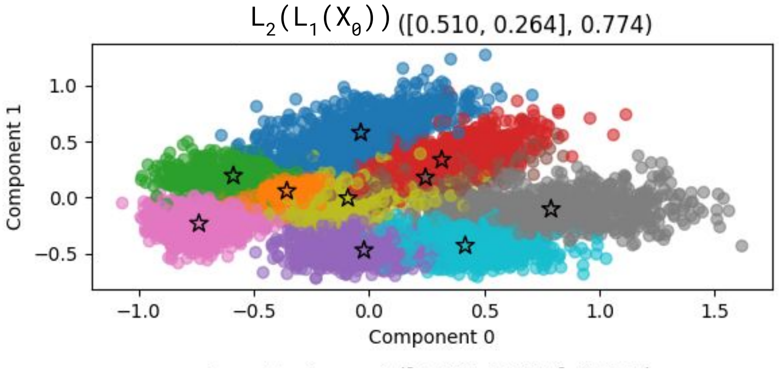


MNIST Model Fault Injection with SliceFI

- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)

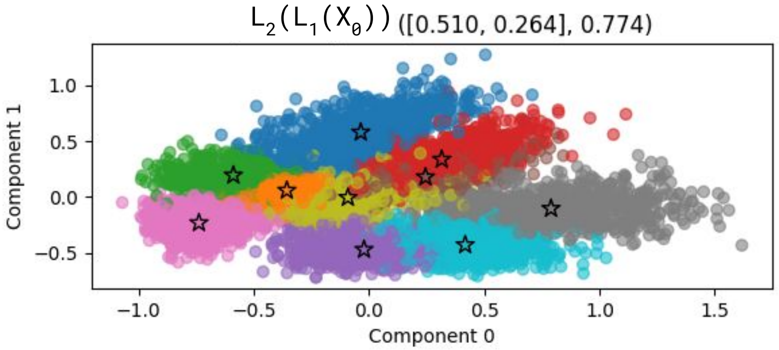
MNIST Model Fault Injection with SliceFI

- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)



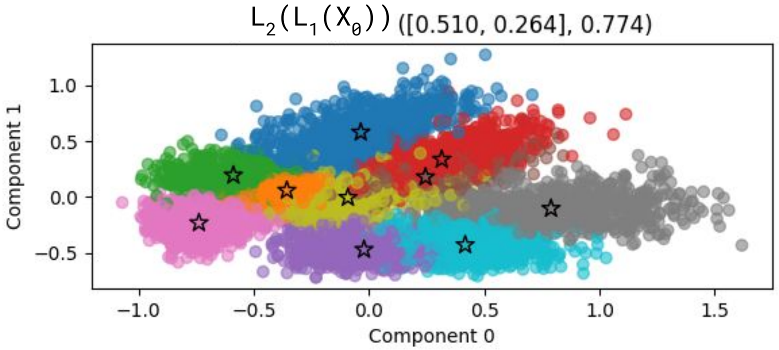
MNIST Model Fault Injection with SliceFI

- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)
  - Full inference:
    - $X_4 = L_4(L_3(L'_2(L_1(X_0))))$



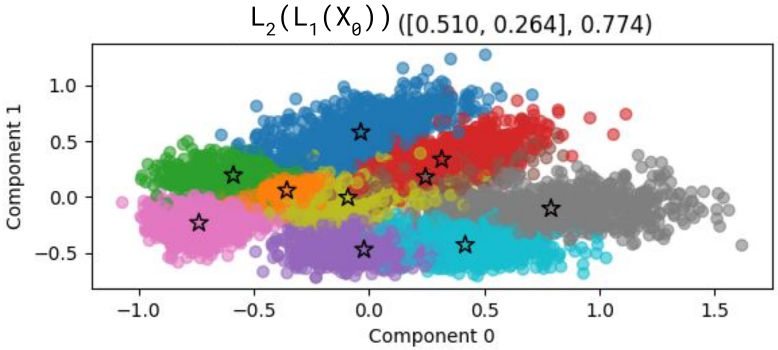
MNIST Model Fault Injection with SliceFI

- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)
  - Full inference:
    - $X_4 = L_4(L_3(L'_2(L_1(X_0))))$
  - SliceFI Inference:
    - $X_4 \sim \mathrm{Heuristic}(L'_2(X_1))$
    - $L_1(X_0)$ is already cached as X_1



MNIST Model Fault Injection with SliceFI

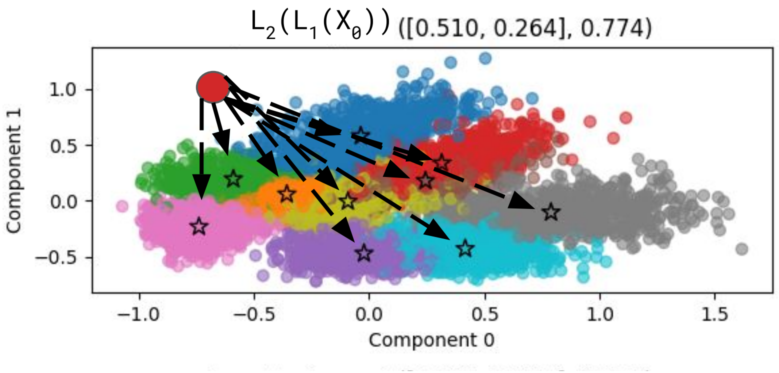
- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)
  - Full inference:
    - $X_4 = L_4(L_3(L'_2(L_1(X_0))))$
  - SliceFI Inference:
    - $X_4 \sim \mathrm{Heuristic}(L'_2(X_1))$
    - $L_1(X_0)$ is already cached as X_1



- Heuristic:
  - Compute the distance between new faulty coordinate and the expected centroids → `[f_dist_c0,...,f_dist_c9]`

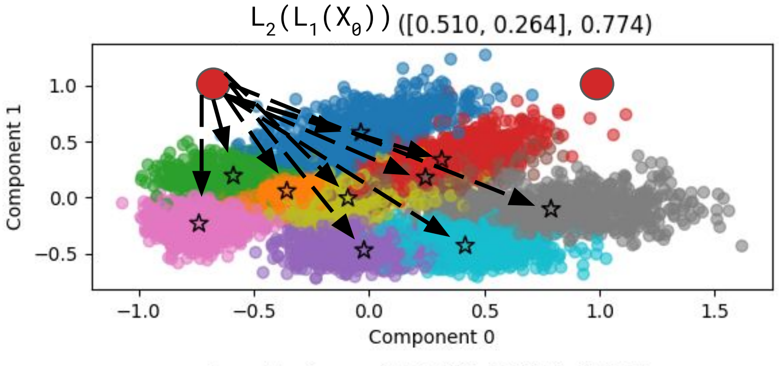
MNIST Model Fault Injection with SliceFI

- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)
  - Full inference:
    - $X_4 = L_4(L_3(L'_2(L_1(X_0))))$
  - SliceFI Inference:
    - $X_4 \sim \text{Heuristic}(L'_2(X_1))$
    - $L_1(X_0)$ is already cached as X_1
- Heuristic:
  - Compute the distance between new faulty coordinate and the expected centroids → `[f_dist_c0,...,f_dist_c9]`



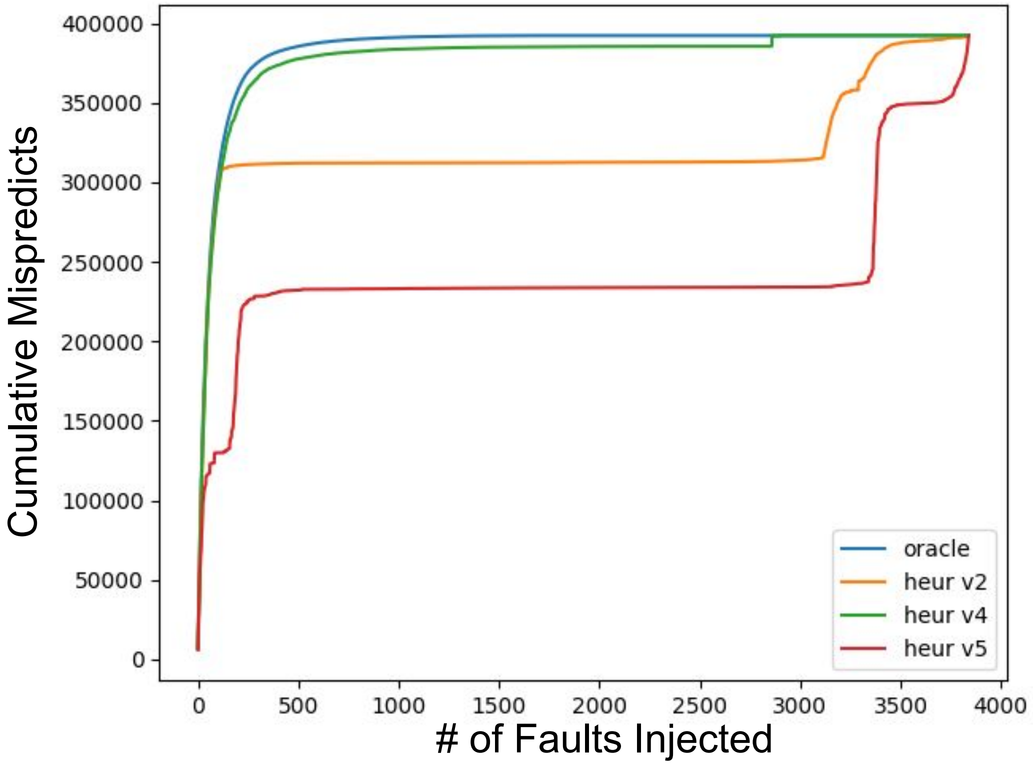
MNIST Model Fault Injection with SliceFI

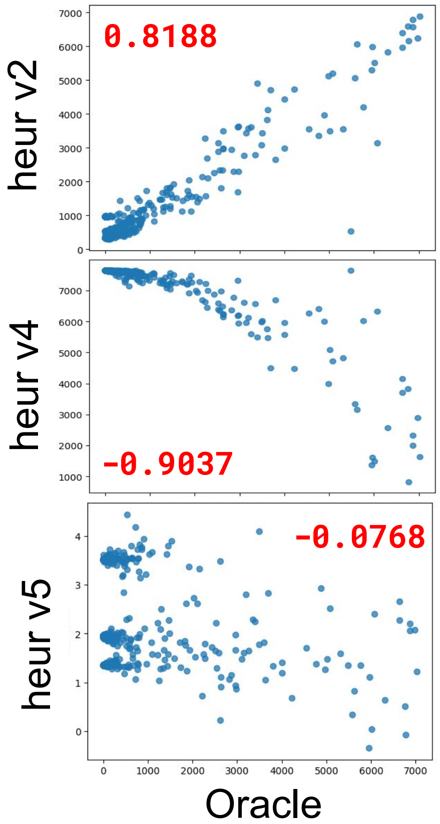
- Imagine we are injecting a fault (single bit flip) in Layer 2 ($L_2 \rightarrow L'_2$)
  - Full inference:
    - $X_4 = L_4(L_3(L'_2(L_1(X_0))))$
  - SliceFI Inference:
    - $X_4 \sim \text{Heuristic}(L'_2(X_1))$
    - $L_1(X_0)$ is already cached as X_1
- Heuristic:
  - Compute the distance between new faulty coordinate and the expected centroids → [f_dist_c0,...,f_dist_c9]
  - Store the difference between those faulty distances and the expected centroid distances (pre-computed)



MNIST Model SliceFI Results

- Heuristics that better resemble the oracle are better at approximating the number of mispredicts caused by injecting a particular fault





SliceFI Summary

- An FI tool that approximates full faulty inference from a faulty model layer
- SliceFI slices a model into individual layers (slices) to speed up FI campaigns
- Preliminary results indicate this approach is feasible for edge neural networks performing classification tasks
- On-going/Future Work:
  - Explore additional heuristics
  - Evaluate effectiveness on regression models
  - Explore whether additional computations can be removed from FI loop
  - Evaluate speed up with SliceFI's additional parallelism overhead

$X'_4 = L_4(L_3(L_2(L'_1(X_0))))$ → SliceFI Approach → $X'_4 \sim \mathtt{Heuristic}((L'_1(X_0)))$

Thank you! Questions?

UC San Diego

Fermilab