Specified Backup for Fragile Parts of LLM

Abhi Morumpalle, Allen Zhang Arnav Marda, Jeffrey Kwan, Harry Qian
UCLA

ABSTRACT

In the past few years, there has been an explosion of interest in Large Language Models (LLMs) for a variety of practical applications. Much of this explosion has been driven by the invention of the Transformer architecture [7]. However, the Transformer architecture inner workings largely remain a mystery. Combining this with the applications that LLMs are finding in the real-world, there a variety of new security risks that these LLMs open up their users to. In this paper, we analyze the robustness of LLMs to random bitflips in the variables, pinpointing specific parts of the LLM that are vulnerable to these hardware errors. We also evaluate the effect that value errors have on efficacy of specific components of the LLM when evaluated against various LLM benchmarks.

1 INTRODUCTION

In the past few years, there has been an explosion of interest in LLMs with the creation of widely available resources like OpenAI's ChatGPT and Meta's open source Llama. Much of the explosion has been driven by the creation of the transformer architecture, which has made a dramatic difference throughout AI, but particularly in the world of LLMs. However, our fundamental understanding of how these objects remains shrouded in mystery.

Because of our lack of understanding of how these objects work, and the quick assimilation of these products into our daily lives, there are a variety of novel security risks that we are being introduced to. One specific error is not so common, but still of practical relevance, is a hardware failure in which a random bitflip occurs in the parameters of our model. An error of such a fashion could have drastic effects on our output, ranging from making the outputs gibberish to outright wrong.

In this paper, we analyze how injecting bit errors into specific locations of a transformer and the LLM model as a whole affect the output. To this end, we use the Mistral-7B [4] as a base model to test on. To inject errors into our base model we use an extension of the PyTEI package introduced in [5]. To quantify the effect of our errors, we compare the score of our base model to the score of the models with errors injected using the PyTEI package with the MMLU framework supplied by DeepEval to evaluate.

Our basic workflow is outlined in the below diagram

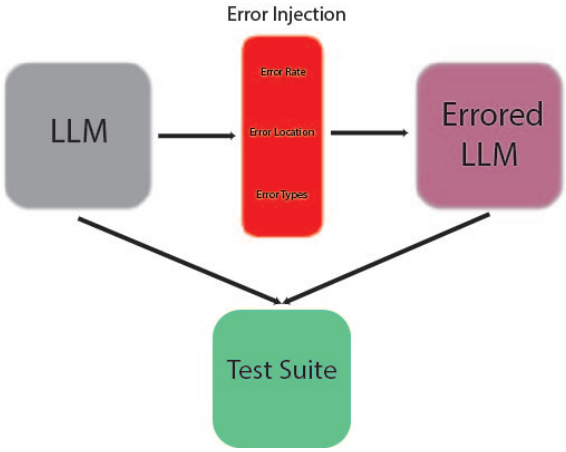

Figure 1: General evaluation workflow for LLM

2 BACKGROUND AND RELATED WORK

There is a related paper [5] that analyzes a model known as a recommendation system. In their work, they build the PyTEI package for injecting models. However, recommendation models differ significantly from LLMs so the overall effect could be quite different for the same error injections.

However, the paper does not do any evaluation on the different of hardware errors in specific parts of the recommendation system, so the question of whether particular parts are more vulnerable is still open.

An interesting part of this paper is the evaluation of possible mitigations against hardware flips, which can also be evaluated in the context of LLMs. In addition, their evaluation had limited scope, and it could be interesting to expand on their analysis by testing a wider variety of errors and examining the tradeoffs of each.

[2] performed resilience analysis on transient errors in logic components, but their framework is highly inefficient in the context of PyTorch, because they employ frequent to-and-back type conversions as PyTorch doesn't natively support bit operations in float tensors. [6] estimated DNN accuracy under transient faults and proposed a Monte Carlo-based estimation method. However, their analysis is also limited to transient errors and do not consider logic / data path errors which are permanent and more impactful.

3 THREAT MODEL.

An attacker's goal is to cause information disclosure or cause a denial of service via system crash or significantly degrading the performance of the LLM. They may attack components such as edge AI chips and accelerators, CPUs and GPUs in LLM training, memory systems storing model weights, and data transfer channels between hardware components. To induce hardware errors, they could somehow achieve write access to memory systems, or

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

physical access to create fault-heavy environments (e.g. radiation source).

3.1 Case Studies

We present 2 case studies, relating to information disclosure and denial of service respectively. However, for the purposes of this paper, we will investigate the effects of the *denial of service* attack objective, specifically performance degradation only.

3.1.1 Case 1 (Information Disclosure). An LLM is running on an edge device (like a smartwatch). The attacker jailbreaks the device, gaining write access to hardware, and inject bit errors with a probability p. The attacker then monitors the LLM output through API calls, observing that the output is NaN XX% of the time. The attacker thus concludes that the LLM has YY number of parameters (see 6.1 for more details).

3.1.2 Case 2 (Denial of Service). A physics research lab is using LLMs for running simulations of their experiments. An attacker somehow manages to convince the lab scientists to house the servers the LLM is running on in their reactor chamber, a source with high radiation. This environment causes many bit flip errors, causing frequent faulty outputs from the LLM.

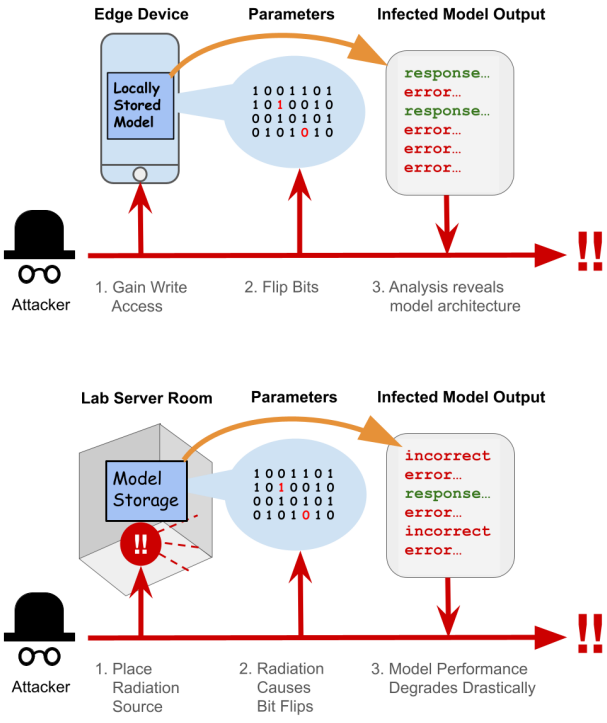

Figure 2: Threat models for case studies: Information Disclosure (top) and Denial of Service (bottom)

4 OVERVIEW OF THE DESIGN

We chose Hugging Face's implementations of Mistral-7B [4] as our model of choice. We evaluated our models on [3], an open-source LLM benchmark, specifically the computer science and astronomy tests that have the injected LLM answer multiple choice questions. For each model with varying error rates, the score is computed as the proportions of correct answers.

4.1 Design Choices

We originally chose GPT2 but noticed a lot of NaN outputs. See 6.1 for analysis. Hence, we switched to an error model that injects *value* errors instead.

We also observed that GPT2 performance on the MMLU benchmark was equivalent to random guessing, which would not be informative of performance drops. We upgraded to Mistral-7B which has a 0.58 average accuracy.

4.2 Design Approaches

4.2.1 Sequential Injection. To analyze the impact of errors in sequential layers on model accuracy, we grouped the decoder layers into sets of four. This grouping reduced computational overhead while still enabling the modeling of sequential error injection behavior. Weights within each group were tracked, and errors were injected with varying probabilities ranging from 10^-9 to 10^-1. For each error likelihood, the model's accuracy on the MMLU benchmark was recorded.

4.2.2 Functional Injection. In order to determine if, and what, components of an LLM (i.e Attention Mechanism, Embedding, etc) were less resistant to SDC, we chose to inject errors into weights associated with these components. For each component's weights, errors were injected with a varying probability from 10^{-9} to 10^{-1}. For each likelihood of errors, model accuracy on the MMLU benchmark was recorded.

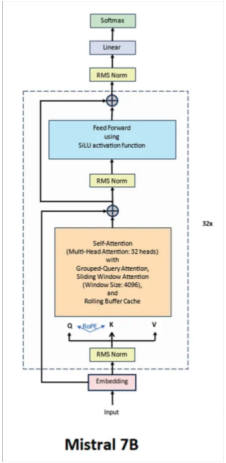

Figure 3: Injected Model's Architecture

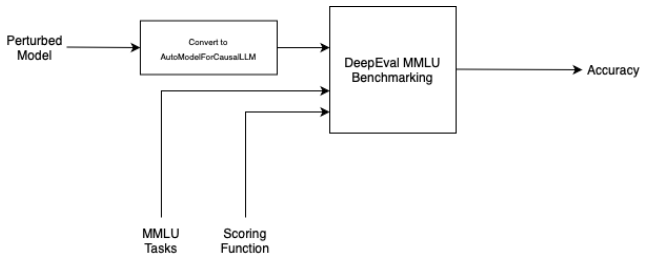

Figure 4: Evaluation Pipeline

5 METHODOLOGY/DESIGN

5.1 Error Injection

In order to simulate bit flips, we load the model parameters, inject errors into them, save the injected weights, and load the new weights into a perturbed model.

To achieve this, we use PyTEI, which provides a flexible API that can generate bit error maps for each part of the model parameters. First, we generate bit masks for all parameters we wish to inject into, with one boolean value for each bit in the parameter when stored as IEEE floats. Then, we specify the probability of a simulated SDC error occuring in the affected parameters, and using PyTorch's implementation of DropOut to efficiently zero out most bits according to the error rate. The mask is then XOR-ed with the stored parameters, where each 1 bit in the mask will trigger a bit flip.

In later experiments, we extended the functionality of PyTEI by creating an API for injecting value errors. A value error occurs when the entire float is replaced with a pre-specified incorrect value. Each float in the parameter tensor is still corrupted independently with a specified probability.

5.2 Evaluation

5.2.1 MMLU Benchmark. We evaluate our perturbed models on the Massive Multitask Language Understanding (MMLU) Benchmark. The MMLU Benchmark contains 15908 questions over 57 subjects. The difficulty of the questions ranges from an elementary level to an advanced professional level[3].

For our particular evaluation framework, we narrowed the scope of our evaluation to the High School Computer Science and Astronomy tasks. The scope was narrowed from 57 tasks to 2 tasks due to the excessive computational costs required to run 15908 instances of inference on each perturbed model. These particular tasks were chosen since they provide around average performance (5) without exceeding the input token limit for GPT-2 and Mistral 7B.

The evaluation was conducted in a 1-shot learning setup to isolate the effect of the perturbation on the model's intrinsic knowledge retention and generalization capabilities from few-shot learning.

5.2.2 DeepEval Testing Framework. In order to standardize the evaluation of the perturbed models, we used an existing library in Python called DeepEval, an open-source toolkit for LLM evaluation. The framework consists of an inbuilt class to run evaluation on particular tasks of the MMLU benchmark.

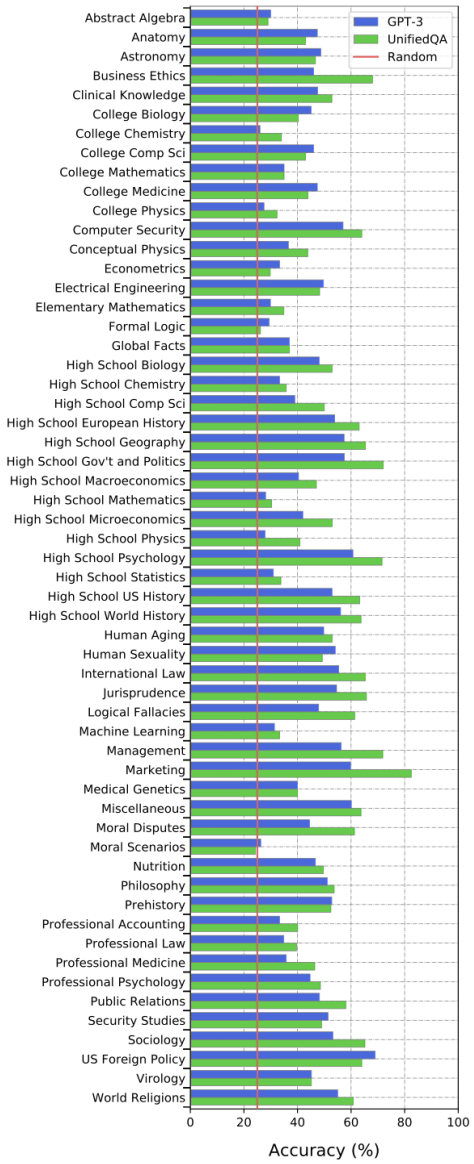

Figure 5: Performance of GPT-3, UnifiedQA and random guessing on all 57 MMLU tasks[3]

The MMLU Benchmarking class takes as input a model, and a set of tasks to run. It then uses the following algorithm to evaluate the model on the benchmark:

In 1, the function `format_input` uses a specific template to format the input to instruct the LLM to correctly answer the question in a specific format. Also, `model.generate()` refers to a generic function to generate output tokens from an LLM given some set of input tokens.

5.2.3 Exact Match Metric. For this evaluation framework and benchmark, we use an exact-match metric. This is a popular metric that outputs a binary result where the input is penalized if it is not an exact match to the target. In this case, we compare the output

```
Input: model, tasks, score_fn
Output: accuracy
Function evaluate(model, tasks, score_fn):
    total_qs ← 0;
    total_correct ← 0;
    for task in tasks do
        golden ← dataset[task];
        for g in golden do
            input ← format_input(g);
            result ← model.generate(input);
            score ← score_fn(result, g.answer);
            if score then
                total_correct ← total_correct + 1;
            end
            total_qs ← total_qs + 1;
        end
    end
    return total_correct / total_qs;
```

Algorithm 1: Evaluation algorithm for model evaluation on MMLU

token(s) to the target, and if it is not an exact match, we return `false`.

5.3 Evaluation Framework

(1) Download HuggingFace model to test on
(2) Fix an error parameter p and specific labelled sections of the model to inject errors into
(3) Inject value errors with each value having probability p of being an error, where our error was replacement with random value from $(0, 1)$
(4) Put our error injected model into a pipeline that conforms to the testing framework (DeepEval MMLU) in our case
(5) Run the tests and evaluate accuracy

5.3.1 HuggingFace Model (Mistral 7B). For the purpose of our experiment, we chose to use the model Mistral-7B [4] from HuggingFace and ran it using the P100 GPU provided by Kaggle. Mistral-7B was chosen for its efficacy at relatively low memory cost. Our baseline score on our MMLU testing cases was 0.6, which is enough to be able to claim a significant difference when injecting errors. This is compared to to GPT-2, in which even larger versions only perform at slightly better than random guessing.

A big benefit of using HuggingFace is that all layers are labelled appropriately, which makes injecting errors into specific components much easier, as we will explain later. Mistral-7B's core is made up of 32 transformer layers, each with the usual self attention, normalization, etc components.

As an aside, to be able to run on the given GPU's memory, we downloaded the model with float16 quantization. We considered further quantization but the error injection became much more complex in this case.

5.3.2 Error Parameters. There are two parameters that we tried changing. An error parameter p and the set of components that we wanted to inject errors into. The error parameter p is the probability that a random value p changes into a random value from $(0, 1)$. This process is run on every component that we want to inject errors into.

The reason we chose a random value from $(0, 1)$ is multi-fold. For one, for this specific Mistral-7B model all weights are between $(0, 1)$ so we were not losing too much by restricting to this. The second, more important reason is that by flipping random bits, the float16 tends to explode to an extremely large value or have NaN. NaN meant that we received errors, whereas large float16 tended to overflow and also throw an error in our code. For this reason, in order to get any results at all, we decided to stick to replacing our small weights with other small weights.

The specific error parameters that we tested were to split up all 32 layers of Mistral-7B into sublayers of size 4 (so 1-4, 5-8, etc) and inject errors into those specific sublayers. We varied p logarithmically from 10^{-9} to 10^{-1}. We also looked into injecting errors into components of each layer, so we tried injecting into all self-attention layers, normalization layers, etc with the same varied values of p.

6 EVALUATION

6.1 NaN Analysis

In this section, we provide an analysis of the probability that the LLM outputs NaN.

Let F be an LLM model with n parameters and x be the input to the model. We would like to find $Pr(F(x) = NaN)$.

Suppose a bit is flipped with probability p.

Let X be the value of a model parameter and X^* be its perturbed value. Note that parameters are iid $\sim Uniform(0, 1)$ due to the design of LLMs [do you have a good source for this allen]. According to [1], a NaN value has all 1s in the exponent field, and a nonzero mantissa. Since $0 \leq X \leq 1$, the exponent field is 01111111 (due to the +127 offset), and suppose there are j 1s in the mantissa.

Then, $X^* = NaN$ if the remaining exponent bit flipped, the rest of the exponent bits don't flip, and not all j mantissa bits turn to 0. Thus,

$$\begin{aligned} Pr(X^* = NaN) &= Pr(\text{only one exponent bit is flipped}) \\ &\quad \cdot (1 - Pr(\text{all mantissa bits are 0})) \\ &= p(1-p)^7 * (1 - p^j(1-p)^{23-j}) \end{aligned}$$

We can represent the value of mantissa as a random variable $\sim Binom(23, 0.5)$. Thus,

$$\begin{aligned} Pr(X^* = NaN) &= \sum_j^{23} Pr(X^* = NaN \mid X \text{ has } j \text{ mantissa 1s}) \\ &\quad \cdot Pr(X \text{ has } j \text{ mantissa 1s}) \\ &= \sum_j^{23} p(1-p)^7 * (1 - p^j(1-p)^{23-j}) \binom{23}{j} 0.5^j 0.5^{23-j} \\ &= 2^{-23} p(1-p)^7 \left(\sum_{j=0}^{23} [1 - p^j(1-p)^{23-j}] \binom{23}{j} \right) \\ &= 2^{-23} p(1-p)^7 \left(\sum_{j=0}^{23} \binom{23}{j} - \sum_{j=0}^{23} p^j(1-p)^{23-j} \binom{23}{j} \right) \\ &= 2^{-23} p(1-p)^7 (2^{23} - 1) \\ &= p(1-p)^7 (1 - 2^{-23}) \end{aligned}$$

Now putting everything together, we note that the model will output NaN if any of the parameters of the model are NaN, due to NaN propagation. Thus,

$$\begin{aligned} Pr(F(x) = NaN) &= Pr(\text{at least one parameter is NaN}) \\ &= 1 - Pr(\text{all parameters are not NaN}) \\ &= 1 - \prod_{i=1}^{n} Pr(X_i \neq NaN) \\ &= 1 - Pr(X^* \neq NaN)^n \\ &= 1 - (1 - Pr(X^* = NaN))^n \\ &= 1 - (1 - p(1-p)^7(1 - 2^{-23}))^n \end{aligned}$$

For $p = 10^{-9}$ and $n \approx 130 * 10^6$ (GPT-2), we get $Pr(F(x) = NaN) \approx 0.1$, and for $p = 10^{-8}$, we get $Pr(F(x) = NaN) \approx 0.7$, which explains why at a low error rate, the model still outputs NaN quite consistently.

6.2 Variation by Layer

6.2.1 Probability Variation. We began by analyzing, at each level, how variations in probability effect the evaluation on the MMLU benchmark at various probability from $1e-7$ to $1e-3$.

As mentioned before, for computational reasons we grouped up layers into groups of size 4. For ease of viewing this result, we only present the value for layer groups 1, 2, 7 and 8. A similar pattern holds for all unshown groups.

Below are our results for each layer group with various probabilities.

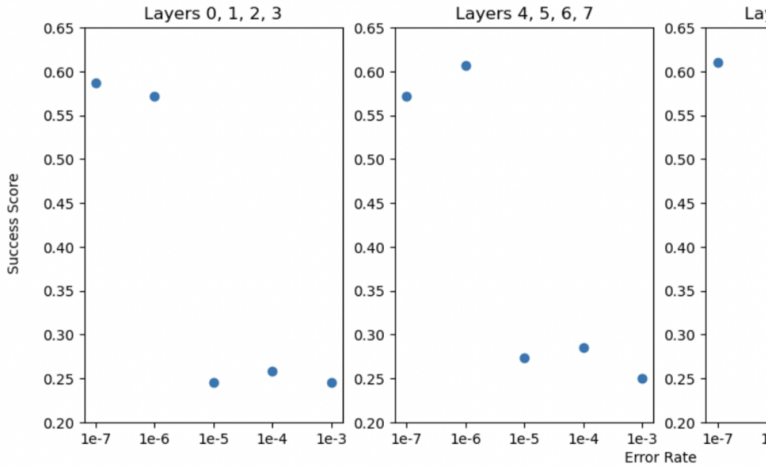

Figure 6: Score when Varying Probability

As one can notice, for probabilities $1e-6$ (this is about 200 total errors for each layer group) there is no noticeable negative effect that our injected errors have on the model.

On the other hand, for probabilities $1e-4$ (about 20000 total errors), the output of our LLM for each layer that we have is essentially no better than random guessing, no matter which layer we have injected into.

The only layer where something interesting seems to be happening is at the $1e-5$ parameter, so we look into this further in the below section.

6.2.2 Zooming in on 1e-5. There are a few interesting things to notice here.

The first is that there does seem to be an effect of the layer on the error injection. For example, errors later layers seem to have less of a degratory effect on the efficacy of the LLM compared to earlier layers. This is certainly an interesting result that seems to suggest that errors in earlier layers propagate to later layers.

Another interesting thing to note is that many there does seem to be a large amount of variation. We will explore this in a later section, but for now it suffices to say that this is because the variation in our error injection process means that some more vulnerable components within the individual layers also have a large effect on the total degradation of the LLM.

6.3 Variation by Component

6.3.1 Probability Variation. We begin by analyzing, for each LLM component (i.e Attention Mechanism, Embedding, etc) how

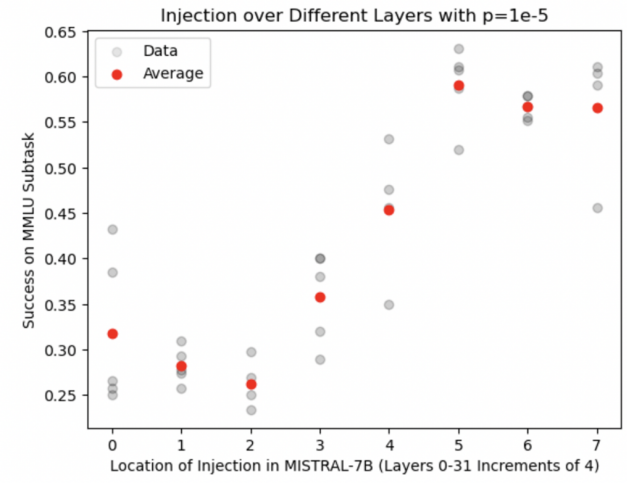

Figure 7: Score when Varying Layers with p = 1e-5

variations in the likelihood of SDC, from 10^{-9} to 10^{-1} effect the model's accuracy. Below are these results.

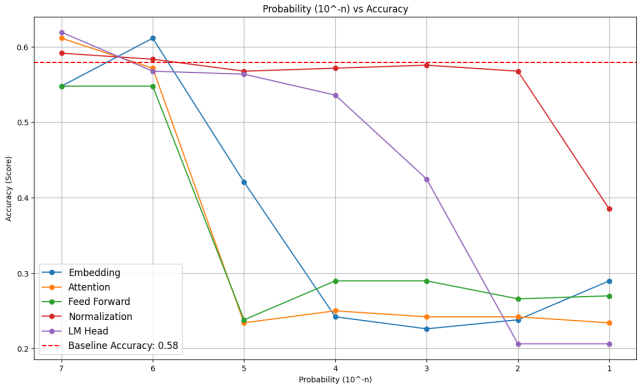

Figure 8: Likelihood of SDC vs Model Accuracy

From the above results, it is evident that certain components are more sensitive to SDC in terms of model accuracy and performance. To quantify this sensitivity, we calculate the **Relative Change in Accuracy**.

$$\text{Relative Change in Accuracy} = \frac{\text{Accuracy}_{\text{baseline}} - \text{Accuracy}_{\text{faulty}}}{\text{Accuracy}_{\text{baseline}}}$$

Below is a graph of this metric for every likelihood.

This visualization shows us how each component, at each likelihood of SDC, degrades in performance relative to the baseline accuracy of the model. Evidently, components like Normalization and LM Head are much more resilient to SDC than the Feed-Forward and Attention components within the decoder layers of the model.

Furthermore, the thresholds at which model performance degrades by 50% across components are visualized below:

To develop a comprehensive sensitivity metric for each component, considering all SDC likelihoods, we compute the **Average Weighted Relative Change in Accuracy Loss (AWRCL)**.

$$\text{AWRCL} = \frac{\sum_{i=1}^{n} w_i \cdot \left(\frac{\text{Accuracy}_{\text{baseline}} - \text{Accuracy}_i}{\text{Accuracy}_{\text{baseline}}} \right)}{\sum_{i=1}^{n} w_i}$$

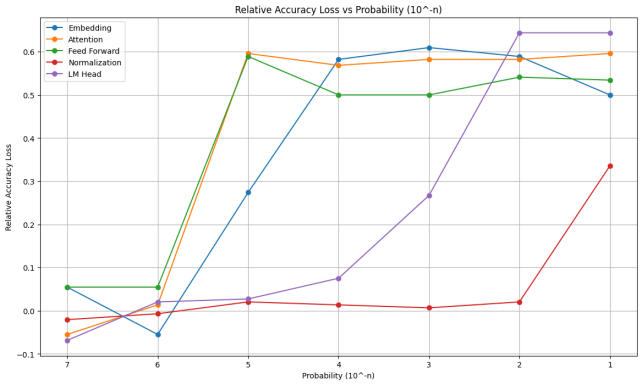

Figure 9: Likelihood of SDC vs Relative Change in Accuracy Loss

images/threshold.png

Figure 10: Thresholds for Model Performance to Degrade Model Performance by 50%

The ranking of LLM components by this metric is displayed below.

Component	AWRCL
Normalization	0.0651
LM Head	0.2797
Embedding	0.4167
Feed Forward	0.4532
Attention	0.4897

Figure 11: Ranking of Components by Average Weighted Relative Change in Accuracy Loss with Respect to SDC Likelihood

These results reveal that **Normalization** and **LM Head** components are the most resilient to SDC. Notably, the LM Head, responsible for mapping vector representations to vocabulary probabilities, shows remarkable resistance despite its outputs being terminal. This aligns with the general observation that errors in earlier layers have a more pronounced impact on model accuracy. The unexpected resilience of the Normalization component, however, warrants further investigation.

6.3.2 Normalizing for Component Size. While the previous analysis compares accuracy degradation by SDC likelihood, components with fewer parameters naturally experience fewer total corruptions. To address this, we normalize the results by the number of parameters in each component, as shown below.

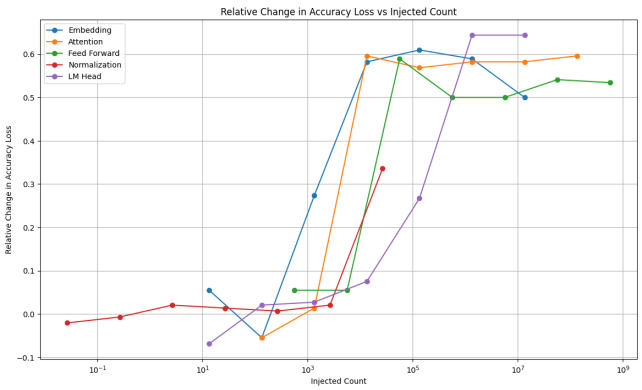

Figure 12: Number of Injected Errors vs Relative Change in Accuracy Loss

The normalized rankings of components, accounting for the number of injected errors, are presented here:

Component	AWRCL
Normalization	0.0193
Feed Forward	0.5370
Attention	0.5820
Embedding	0.5905
LM Head	0.6042

Figure 13: Ranking of Components by Average Weighted Relative Change in Accuracy Loss with Respect to Injected Error Count

The results suggest that, excluding the Normalization component's AWRCL, the sensitivity of components to SDC becomes roughly similar when normalized for parameter count. This finding highlights that smaller components require a proportionally larger fraction of errors to experience equivalent accuracy degradation compared to larger components. Consequently, smaller components inherently exhibit greater resilience to SDC.

In real-world scenarios, assuming uniform bit error rates across memory, the proportion of errors across components is expected to be similar. Thus, these findings offer practical insights into resource-constrained environments, where prioritizing protection (e.g., using ECC or activation clipping) for specific components is critical.

6.3.3 Formula to Model Accuracy Degradation. Through our results, and analysis, we aim to provide a formula to model how sensitive a component is to accuracy degradation as a result of SDC, by identifying key factors.

$$D \propto N \cdot S_c \cdot P_f \cdot A \cdot (1 - R)$$

Where:

D : Performance degradation,

N : Number of parameters or operations in the component,

S_c : Sensitivity factor for the component, measured by the AWRCL

P_f : Fault likelihood (probability of a fault per parameter or operation),

A : Amplification factor,

R : Redundancy factor,

This formula captures the relationships derived from our experiments with sequential layer injections and component-level injections. Notably, the **Amplification Factor** A encapsulates the tendency for earlier layers to propagate faults more significantly through the model. Conversely, the **Redundancy Factor** R represents the degree of overlap in calculations, typically higher in larger components.

In the case of the sparse Mistral 7B model, reduced redundancy contributes to greater sensitivity in larger layers, providing additional context to our earlier observation that larger components are generally less resilient to SDC.

7 DISCUSSIONS

There are still many possible things that one could try. As a group, we were heavily restricted by our computational resources, so with more computation resources and more time it would be possible to try many more different things, as well as trying some more state of the art models with many more parameters than Mistral-7B.

It could be interesting to try inject errors into specific sections of each layer (such as attention or normalization), instead of just all the layers and see or all sections in a particular layer. We have confidence that this could give interesting results because of the high variation in some of our injection parameters, which suggests that where our random errors actually are injected into even with our given parameters is also important.

It would also be interesting to look at the performance degradation on various other benchmarks such as translation, coding capabilities, or reasoning tasks.

Our research is also limited in the sense that we looked at value injections. As a whole, silent bit errors seem to be completely catastrophic for some standard machine learning implementation. For example, in the usual case where the weights lie between 0 and 1, we've shown in 6.1 that NaN is likely for any large amount of bit errors. In float16 representation, the errors tend to cause overflow which broke the evaluation framework that we were working in. The final case is int4 quantization. Here, bit errors actually tend

to be fine. In addition, random bit errors here are close to random value errors, which justifies our choice to look at random value errors instead.

8 CONCLUSION

The large takeaway is that all value errors are certainly not equal. It seems that errors in earlier levels tend to degrade the power of our LLM significantly more than errors at later levels. In addition, it seems that certain parts of the LLM tend to be more resistant to errors (such as the batch-normalization layer). In addition, we have shown that, in general, random bit errors in an LLM's weights tend to be catastrophic in the sense they have a high probability of completely breaking the LLM.

The limitation in our work is in connecting these two cases. It's unknown if our small value errors are a good model for bit errors that do not completely break the LLM. Hence, it's unknown whether large variation errors will have a larger degrading effect than the ones we have injected. In addition, it is still unknown if our results on error degradation for specific layers still hold for the larger case of completely random bit errors.

More research is needed for all the above mentioned cases to draw stronger conclusions on the security of LLM's to random bit flips.

9 CONTRIBUTIONS OF EACH TEAMMATE

Abhi worked on the notebook for injecting errors by component, performed the runs for this experiment on Kaggle, and performed the analysis of the results. Abhi also modified the error injection library to output the number of injected parameters. Abhi wrote sections **4.2 Design Approaches** and **6.3 Variation by Component** of this report.

Allen developed the notebook for running the experiments and ran runs of the experiment on Kaggle. He wrote sections **1 Introduction**, **5.3 Evaluation Framework**, **6.2 Variation by Layer**, **7 Discussions** and **8 Conclusion**.

Jeffrey implemented the initial code for LLM inference, and ran some of the experiments. He also wrote sections **3 Threat Model**, **4.1 Design Choices**, **6.1 NaN Analysis** and part of **2 Background and Related Work**.

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] Yi He, Prasanna Balaprakash, and Yanjing Li. 2020. Fidelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 270–281. https://doi.org/10.1109/MICRO50266.2020.00033

[3] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. (2021). arXiv:cs.CY/2009.03300 https://arxiv.org/abs/2009.03300

[4] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. (2023). arXiv:cs.CL/2310.06825 https://arxiv.org/abs/2310.06825

[5] Dongning Ma, Xun Jiao, Fred Lin, Mengshi Zhang, Alban Desmaison, Thomas Sellinger, Daniel Moore, and Sriram Sankar. 2023. Evaluating and Enhancing Robustness of Deep Recommendation Systems Against Hardware Errors. (2023). arXiv:cs.IR/2307.10244 https://arxiv.org/abs/2307.10244

[6] Abhishek Tyagi, Yiming Gan, Shaoshan Liu, Bo Yu, Paul Whatmough, and Yuhao Zhu. 2024. Thales: Formulating and Estimating Architectural Vulnerability Factors for DNN Accelerators. (2024). arXiv:cs.AR/2212.02649 https://arxiv.org/abs/2212.02649

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. (2023). arXiv:cs.CL/1706.03762 https://arxiv.org/abs/1706.03762

A OVERFLOW FORM OTHER SECTIONS

Sometime you ware super excited about some details that does not quite fit with the rest of the paper goes here. For example, some details about how you instrumented the Android Linux kernel should go to appendix, and for really curious reader to read. Remember it's appendix, so the reader is not required to read, and you should not put critical information in appendix that is crucial for understanding the rest of the paper.