On the Fault Sensitivity of Natural Language Embeddings Computation

Sumeet Sapkal, Ussama Zahid, Giulio Gambardella, Haralampos-G. Stratigopoulos, Brian Clerkin

On the Fault Sensitivity of Natural Language Embeddings Computation

Sumeet Sapkal*, Ussama Zahid*, Giulio Gambardella*, Haralampos-G. Stratigopoulos†, Brian Clerkin*
*Synopsys, Silicon Innovation Group, Dublin, Ireland
{sapkal, ussama, giuliog, bclerkin}@synopsys.com
†Sorbonne Université, CNRS, LIP6, Paris, France
haralampos.stratigopoulos@lip6.fr

***Abstract*—Transformer-based text embedding models have completely changed the landscape of natural language processing domain resulting in enhanced semantic understandings. They are pivotal to achieve high accuracies for various tasks like classification, semantic similarity, and are integral part of Retrieval Augmented Generation (RAG) pipelines (in conjunction with Large Language Models (LLMs)). Embeddings on natural language entries can be computed at the edge, thanks to the inherent compression they provide before communication to the cloud where LLMs are usually executed, thus their computation is critical in terms of accuracy, power and reliability. This work focuses on the latter, by conducting extensive error sensitivity analysis on the text embedding transformer models, targeting different applications using state-of-the-art models. Interestingly, the error injection campaign identifies that mainly a portion of the self-attention layer, specifically the Value tensor (V), is very sensitive to single bit-flips. These results pave the way for a more effective error tolerance approach in future transformer-based embeddings and possibly other self-attention based models.**

***Index Terms*—Text-Embeddding, Transformers, Semantic Search, Reliability, Hardware Faults, Bit-flips.**

I. Introduction

Retrieval Augmented Generation (RAG) pipelines [1] have revolutionized the way Question-Answering (QA) systems operate by combining the strengths of retrieval-based and generative models. RAG pipelines are pivotal in providing accurate and contextually relevant answers, which is at the core of copilot systems. A key component of RAG pipelines is the semantic search, which is largely dependent on the computation of embeddings. RAG pipelines are a hybrid approach to QA where the system first retrieves relevant documents or passages using a retriever component, aggregates this information to refine the search, and finally generates the answer using a generator component. The process can be broken down into three main stages:

1) *Retriever*: fetches relevant documents or data from an external knowledge database based on the input question.
2) *Aggregator*: refines the retrieved information by applying additional filtering or weighting to focus on the most relevant and contextually significant portions of the retrieved documents.
3) *Generator*: synthesizes a coherent and contextually relevant response using the aggregated information. This part of the pipeline leverages advanced language models (like GPT [2], Llama [3], Claude or Gemini [4]) to generate human-like responses that answer the question comprehensively.

At the heart of the retriever component lies semantic search, which aims to understand the meaning and context behind the words, rather than merely matching keywords. Embedding models play a crucial role in performing semantic search. Embeddings are dense vector representations of words, phrases or documents that capture semantic meaning, making it possible to measure similarity between pieces of text based on how they are represented in a high-dimensional space. Embeddings are typically computed using transformer-based models i.e., Bidirectional Encoder Representations from Transformers (BERT) [5], General Text Embeddings (GTE) or BAAI General Embedding (BGE) [6]. These models are trained to understand contextual relationships between words, phrases, and sentences. The semantic understanding process involves:

- *Contextual Embedding Models*: These models generate embeddings that consider the context in which words are used, ensuring that similar meanings are captured accurately. For example, "watch" in *Let's watch a movie* and *I bought a new watch* will have different embeddings representing action and object respectively.
- *Document Embeddings*: Entire documents or segments can be embedded to create vector representations of larger texts, enabling efficient similarity searches across large collections.
- *Similarity Score*: To measure the similarity between embeddings, cosine similarity is commonly used. This measures the cosine of the angle between two vectors (embeddings), giving values between -1 (less similar) and 1 (more similar).

As the quality and accuracy of the generated answer from the Large Language Model (LLMs) heavily depends on the dependability of the retriever, which can be executed at the edge and possibly in harsh environment, we asked ourself the question: *How would embeddings computation behave in presence of hardware faults?* This topic has recently gained

This work is partially supported by the European Network of Excellence dAIEDGE under Grant Agreement No. 101120726.

popularity after the report [3] identifying how 78% of interruptions during the training of modern LLMs were attributed to hardware issues, with the majority due to Graphics Processing Units (GPUs)-related issues. Several studies have been carried out to answer the similar question on other Machine Learning (ML) models, such as LLMs [7], transformers [8]–[11], Convolutional Neural Networks (CNN) [12], [13] and Spiking Neural Networks (SNN) [14], with all the work trying to address the topic of AI dependability.

II. Background and Related Work

A. GPU

In recent years, GPUs have evolved expeditiously from a single gaming industry to a more popular and common High Performance Computing (HPC) industry [15]–[17]. For a long time, computationally expensive workloads were limited by hardware availability. The exponential growth in computationally heavy tasks across multiple domains, combined with the need to reduce the execution time, has led to far-reaching adoption of GPUs. One such example of tasks is the auto-regressive language modeling using LLMs. Over the years, the number of parameters in LLMs has been dramatically increasing [18] [19], leading to the use of large GPUs farms.

B. GPU reliability assessment

GPUs, like any other hardware device, can suffer from hardware faults, i.e. transient faults (a.k.a. soft errors) caused by cosmic rays or radiation and permanent faults caused by physical defects, aging, and wear-out mechanisms. Especially in safety-critical applications, i.e., autonomous vehicles, medical equipments, etc., where the stakes are very high, a single error could lead to hazardous situations. To this end, given the increased adoption of GPUs, it is essential to assess the reliability and fault tolerance of the application executed on GPUs [20]–[23].

Reliability assessment typically involves fault injection experiments to examine how the system behaves under failures. Such experiments can be performed at the application level by emulating faults into the application, by simulating faults at the micro architectural-level or circuit-level, by direct manipulation of the hardware, or by radiation testing [24]–[27]. For example, there are several software-based application-level fault injection frameworks for neural networks built upon popular machine learning frameworks, such as PyTorchFI [20] and TensorFI [28]. Another possibility is to use fault injection tools to emulate fault effects in the GPU running the application, such as SASSIFI [29], NVBitFI [30], and GPU-QIN [31]. All these tools work at the SASS level, i.e., the low-level assembly language for NVIDIA GPUs. Application-level fault injection offers flexibility and scalability, and as long as the fault model can capture hardware-specific failure modes, it can offer realism too. Of course, the closer the fault injection campaign is to the hardware, the more trustworthy the conclusions are. However, hardware-based fault injection is very slow and computationally expensive for large models and may still not capture all real-world effects.

C. Transformers

Transformers [32] are usually composed of an encoder and a decoder block. The former takes as input a sentence (represented as a sequence of tokens) and outputs a matrix representation of the input, while the latter generates natural language starting from the matrix representation. The primary function of the encoder is to transform the input tokens into contextualized representations, by considering the whole sequence of tokens (or sentence), thus increasing the semantic understanding of the whole sentence, thanks to the multi-head attention layer. Figure 1 shows the internal architecture of a complete layer in the encoder, which is stacked in a sequence of layers (12 or 24 in the examples used in this work) with the same internal structure.

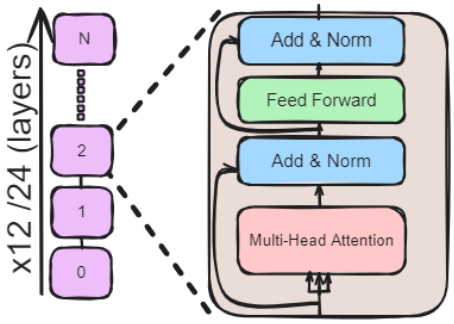


Fig. 1: Transformer Architecture (Encoder)

The core of the computation in each layer is performed in the multi-head attention module, whose internal structure is represented in Figure 2. The block is computed similarly for each head (12 or 16 in the targeted models) which can be done fully in parallel. For each head, the input tensor is split into 3 sub tensors named `Query` (**Q**), `Key` (**K**), and `Value` (**V**). As shown in Figure 2, while the first two undergo a linear layer (dot product usually referred as `Matmul`), `scaling` and `softmax`, the **V** tensor is forwarded to the `context layer` computation directly.

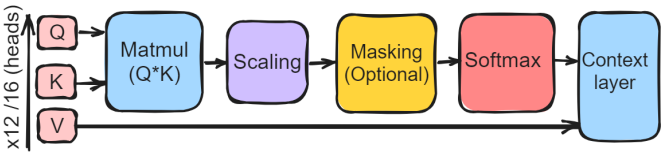


Fig. 2: Inner topology of Multi-Head attention

At the end of the multi-head computation, as shown in Figure 1, results computed by all the heads are aggregated and normalized before going through the feed forward computation, composed of two linear layers with a ReLU activation between them. The output of the final encoder stage is a set of vectors, each representing the input sequence with a rich contextual understanding. In embeddings computation, this is used as a final result, while in generative models, similar but inverse computation is performed in the decoder phase.

D. Transformers reliability

Transformer models did not undergo the level of reliability investigation performed on CNNs [12], [13], but recently the

TABLE I: Model Architecture

Model	Data type	Layers	Heads	Embedding size	# Parameters
stella	Float32	24	16	1024	435 M
gte-base	Float32	12	12	768	137 M

research community has started to study this architecture too. To the best of our knowledge, most existing research has concentrated on image processing, particularly: (1) Vision Transformers, which have been evaluated for soft error vulnerability using fault simulation [8], [10], [11] and radiation experiments [10], with proposed hardening techniques focusing on algorithm-based fault tolerance or preventing the propagation of corrupted values; and (2) Swin Transformers [9], which have also undergone reliability analysis, demonstrating that simply suppressing out-of-range output neuron values was insufficient for fault mitigation, requiring input and weight ranging as well.

E. MTEB

Massive Text Embedding Benchmark (MTEB) [33] has been developed as a dataset repository to compare and rank different text embedding models on multiple tasks. Over time, it grew to be massive, including 8 different tasks and 58 different datasets, and it is considered as the state-of-the-art approach to evaluate text embeddings models, enabling the fair comparison of different models on multiple tasks.

III. Experiment

We conducted experiments on two different models: `stella_en_400M_v5 (stella)` [34] and *`gte-base-en-v1.5 (gte-base)`* [35] as shown in Table I. We selected three tasks on four different datasets from MTEB to analyze if fault sensitivity is model, task or dataset dependent. The selected tasks, their metrics, relative datasets, and their baseline accuracies are listed in Table II.

A. Models and their comparison

Our goal for selecting two different sized models was to study if there is a relation between model size and its fault sensitivity. This setup also allowed us to examine and conduct a comparative study of fault tolerance between a complex and simpler model. As shown in Table I, the *`stella`* (435 million parameters) is a much larger model than *`gte-base`* (137 million parameters). It nearly has double the number of heads and layers making it more suitable for capturing complex relations in input text.

B. Tasks

Since each task has a different application in real-life and each might have its own set of risks, we investigated whether a model is fault sensitive with different tasks.

Semantic Textual Similarity (STS) and Pair Classification tasks are very similar in nature, yet models perform very differently on each task. This reinforces the importance of conducting experiments on different models and tasks to check if the fault sensitivity is task and dataset independent or not.

TABLE II: Tasks and their baseline accuracy

Task	Metric	Dataset	gte-base	stella
Classification	Accuracy	ToxicConversations	83.01	67.96
		Banking77	87.75	88.31
STS	Spearmen Correlation (Cos)	STS12	74.43	75.81
Pair Classification	Average Precision (Cos)	TwitterSemEval2015	75.52	76.69

C. Error Model

The goal of our analysis is to assess error sensitivity independently of the target hardware on which the model will be executed during inference. To do that, we selected single bit flipping on the activation tensors, following the error model also adopted by PyTorchFI [20]. The choice has been made to abstract the evaluation from any hardware specific details (like internal mapping or scheduling), which are required to effectively define more complex multi-bit error models. To assess the fault tolerance of the model, we are injecting errors during the inference execution, running the full testset to compute the target metric. As shown in Table I, we targeted the original model without any quantization, thus error injection is performed on *Float32* datatype. Early phase of the experiments demonstrated how bit-flips in mantissa bits lead to negligible impact, leading the campaign to focus on exponent bits. By flipping the bit in the exponent we shift the targeted value by order of magnitudes causing a significant impact on the output value.

It should be noted that the injection results are not reported for the 7th bit of the exponent, as the bit flipping often results in Not A Number (NAN) representation, following IEEE Standard for Floating-Point Arithmetic (IEEE754), resulting in erroneous execution.

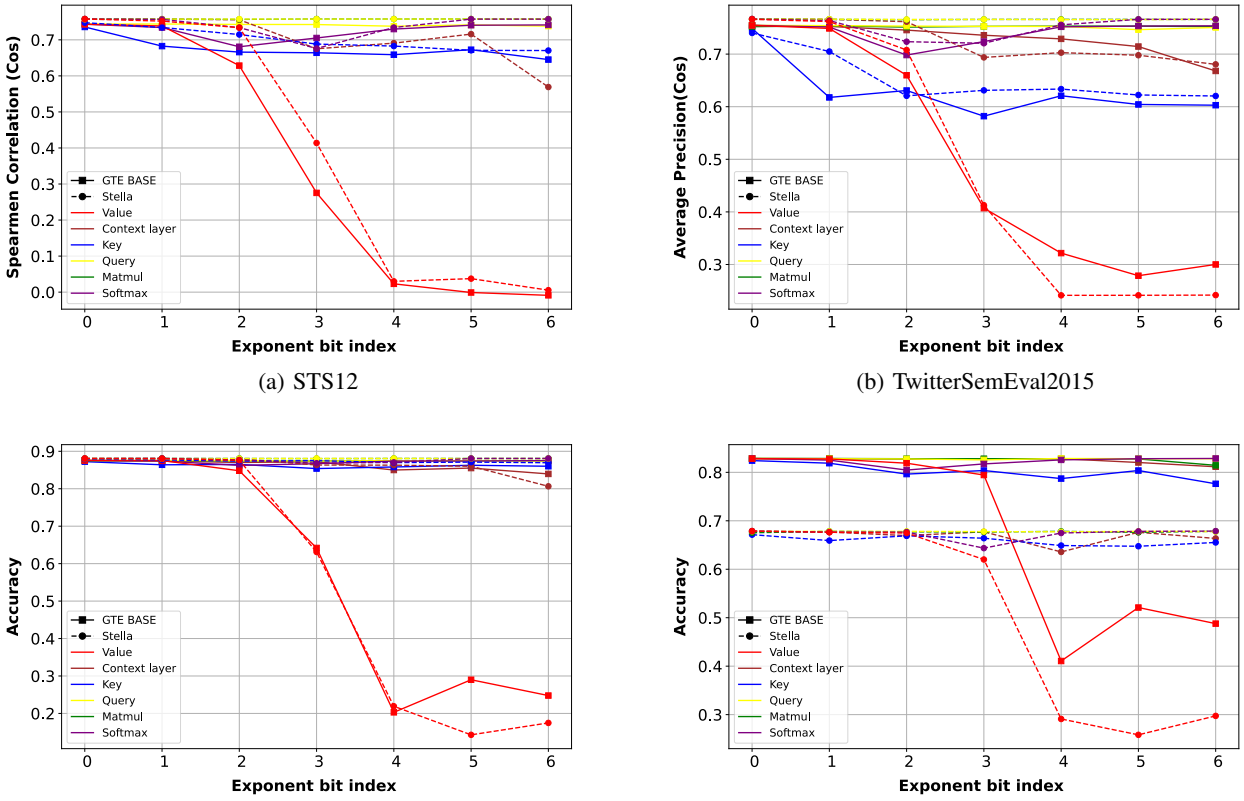
D. Error Injection

The error injection module was developed in the PyTorch framework and it enables us to inject errors using pointers during the runtime of the model. Algorithm 1 describes the overall error injection experiment.

Algorithm 1: Bit-flip error injection evaluation at different token indexes

```
for every tensor in attention module do
    for every experiment do
        for every exponent bit index 1-6 do
            for every head do
                for every layer do
                    Inject error at random token indexes &
                    Run MTEB testset evaluation
                end for
            end for
            Save results
        end for
    end for
end for
```

Using Algorithm 1, we iterate exhaustively over the 12 layers, and each of the 12 heads within a layer for *`gte-base`*, while for *`stella`* we iterate over 24 layers and within each layer 16 heads.



(a) STS12

(b) TwitterSemEval2015

(c) Banking77

(d) ToxicConversations

Fig. 3: Metric fluctuations (Min) score for different bit positions for all tensors across 4 different datasets

We randomly select a token for each tensor and we flip the target bits iteratively along the bit indexes across the exponent. For one tensor, the time required for fault injection experiment is the product of $heads * layers * bits * time$ (on MTEB task) for the complete testset. For example, running the error injection experiment on a single tensor for the `stella` model on the pair classification task requires 24 seconds. So, for a single tensor, the overall experiment takes $24 * 16 * 7 * 24 = 64512$ seconds or 18 hours, therefore the total time required for all tensors would be $18 * 6 = 108$ hours or ~ 4.5 days. We adopted a random token selection strategy during the injection experiments to emulate the hardware faults occurring during the data communication or tokenization.

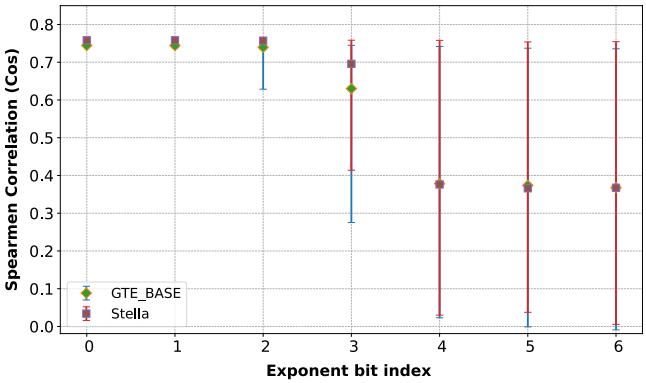
IV. Results and Analysis

To conduct the experiments, we used a NVIDIA A100 GPU, PyTorch 2.3.1, Python 3.9, and CUDA 11.8.

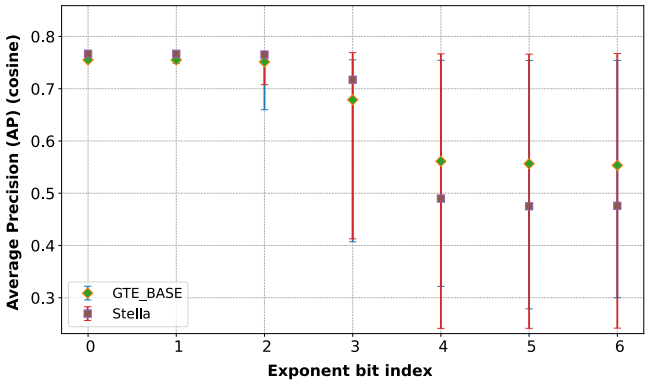
Figure 3 shows the minimum metric scores across all layers and heads for the two models applied to four different datasets, plotted against the bit index where the bit-flip occurs. Each colored curve corresponds to the specific tensor affected by the bit-flip, while the marker of the curve indicates the model.

First evidence shows that **V** is the most sensitive tensor across all models, tasks, and datasets, showing significant drops in accuracies for the most significant bits, ultimately compromising the application. Secondly, the **Q** tensor shows virtually no fault sensitivity in our injection campaign with a maximum drop of 0.87% across all experiments. Similar low fault sensitivity is recorded for `Matmul` (also known as attention scores), which is probably due to the following `scaling` layer limiting the dynamic range of the output. The same assumption can be made to explain the low fault sensitivity observed for the `softmax` layer since this layer performs dynamic range scaling by construction. The sensitivity of other tensors changes appreciably across different tasks. Specifically, for the classification tasks on the Banking77 and ToxicConversations datasets, the change in accuracy is very limited, with a maximum drop of 1.94% in **K**, 1.5% in `softmax`, and 7.6% in `context layer` for Banking77, and 5.38% in **K**, 3.61% in `softmax`, and 4.39% in `context layer` for ToxicConversations. However, for the STS and Pair Classification tasks, tensors **K** and `context layer` show significant impact on accuracy, which for **K** is evident even at low bit positions. The maximum drop is 18.9% for `context layer` and 17.3% for **K**. This is likely due to the nature of these tasks, where the metric relies on comparing two separately computed embeddings, leading to a major impact if one of the two outputs is affected by faulty computation. Given the major impact on accuracy of tensor **V**, we will focus the remaining of the discussion on this tensor.

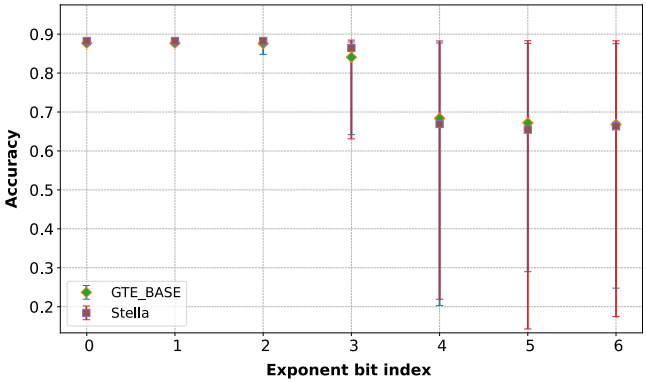
Figure 4 provides more detailed results for tensor **V** using bar plots, showing the mean, maximum, and minimum scores



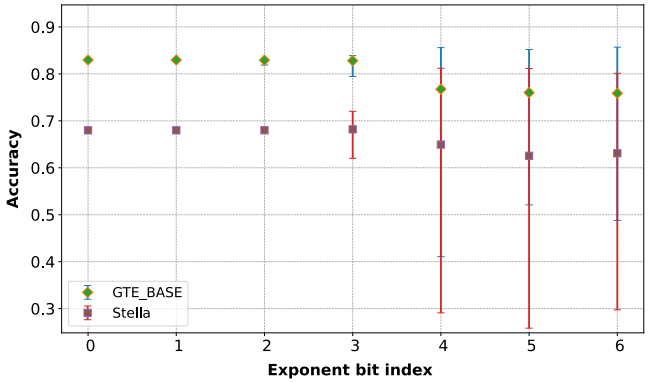
(a) STS12



(b) TwitterSemEval2015



(c) Banking77



(d) ToxicConversations

Fig. 4: The changes in accuracy in the most error sensitive tensor **V**

observed as a function of the bit flip position. As it can be seen, the mean scores have negligible drop until bit 3, whereas from bit 4 onward there is a negative trend. The high sensitivity of **V** is likely due to it being followed directly by matrix multiplication in `context layer`, resulting in error propagation to the multi-head attention output. In the case of single bit flipping operation on the tensor, the original value can shift by orders of magnitude and, as there are no `scaling` or `softmax` layers following, these extreme values are not suppressed and are subsequently passed onto the next layers. The normalization performed at the output of the multi-head attention module is not able to suppress these extreme values since to normalize a tensor it needs to calculate the mean that could be very large for a faulty tensor leading to shift in the distribution, which, in turn, could be further shifted by the skip connections.

The sensitivity of a model to faults affecting tensor **V** depends on the task and the dataset used. As it can be observed in Figure 4, the percentage drop for tasks STS and Pair Classification is much larger compared to the classification tasks, and for classification tasks the percentage drop is larger for the Banking77 dataset compared to the ToxicConversations dataset. Also, for the classification task on the TwitterSemEval2015 dataset, `gte-base` appears to be more fault sensitive compared to `stella`.

TABLE III: Mean, **Max** and *Min* scores for `gte-base` on ToxicConversations

Tensor \ Bit	0	1	2	3	4	5	6
Context	82.97 (**83.10**- *82.86*)	82.97 (**83.15**- *82.81*)	82.96 (**83.25**- *82.71*)	82.96 (**83.15**- *82.71*)	82.96 (**83.10**- *82.81*)	82.97 (**83.20**- *82.03*)	82.94 (**83.34**- *81.15*)
Query	82.97 (**83.10**- *82.86*)	82.97 (**83.05**- *82.81*)	82.97 (**83.10**- *82.76*)	82.96 (**83.10**- *82.56*)	82.97 (**83.10**- *82.86*)	82.97 (**83.10**- *82.81*)	82.97 (**83.20**- *82.81*)
Value	82.96 (**83.10**- *82.81*)	82.96 (**83.15**- *82.76*)	82.92 (**83.20**- *81.88*)	82.79 (**83.88**- *79.44*)	76.74 (**85.59**- *41.06*)	76.01 (**85.20**- *52.09*)	75.87 (**85.69**- *48.77*)
Matmul	82.97 (**83.10**- *82.86*)	82.97 (**83.15**- *82.86*)	82.97 (**83.20**- *82.81*)	82.98 (**83.15**- *82.81*)	82.97 (**83.20**- *82.81*)	82.97 (**83.10**- *82.76*)	82.95 (**83.10**- *81.44*)
SoftMax	82.97 (**83.05**- *82.86*)	82.95 (**83.10**- *82.51*)	82.928 (**83.54**- *80.46*)	82.92 (**84.22**- *81.73*)	82.95 (**83.30**- *82.56*)	82.98 (**83.15**- *82.81*)	82.97 (**83.15**- *82.86*)
Key	82.95 (**83.39**- *82.41*)	82.89 (**83.64**- *81.88*)	82.82 (**83.54**- *79.63*)	82.80 (**84.03**- *80.37*)	82.83 (**84.08**- *78.71*)	82.83 (**83.64**- *80.37*)	82.79 (**83.93**- *77.63*)

The detailed results of the error injection campaign for `gte-base` trained on the ToxicConversations dataset are shown in Table III. The column names from 0-6 represent the bit indexes where the bit-flip occurs and each row represents a tensor. Each cell lists the mean, **maximum**, and *minimum* scores of the model observed across the experiments. Interestingly, while **V** is the most fault sensitive tensor causing the largest drop in scores, a fault in **V** can potentially result in a positive shift in score. For example there is 2.5% increase from the baseline accuracy for a bit-flip at position 3.

TABLE IV: Memory usage and computation for tensor **V** in the multi-head attention module

Model	Memory usage	Computation
gte-base	17.20% (178 kB)	4.3%
stella	16.84% (254 kB)	6.6%

Table IV shows the percentage of memory space usage and computation by tensor **V** within the multi-head attention module. The values depend on the context length and embedding size. We considered the average sequence length (computed across the complete dataset) and batch size as 1 for the calculation. It should be noted that, with longer sequence length, the percentage of memory usage and computation by **V** will decrease. Although the memory usage and computation of tensor **V** is significantly less compared to the whole attention module, it proves to be a critical tensor and should be considered for protection against faults.

V. Conclusions and Future Work

This work proposed an analysis of the fault sensitivity of transformer-based embedding models, by performing an extensive token-based error injection on several applications. Experimental results show that the tensor **V** is the most sensitive to hardware faults for all tested applications causing high impact on the respective metric score. Furthermore, for specific applications such as semantic similarity, other tensors are also fault sensitive, even if with less impact than **V**. Future work will focus on understanding how our findings can be generalized, evaluating more models, tasks, and datasets, trying to correlate applications with most likely sensitive tensors. Furthermore, we will focus on the development of approaches to increase fault tolerance in text embedding inference by means of selectively protecting critical tensors.

References

[1] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 33, Dec. 2020, pp. 9459–9474.

[2] T. B. Brown *et al.*, "Language models are few-shot learners," *arXiv:2005.14165*, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[3] A. Dubey *et al.*, "The llama 3 herd of models," *arXiv:2407.21783*, 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[4] Gemini Team *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv:2312.11805*, 2023. [Online]. Available: https://arxiv.org/abs/2312.11805

[5] J. Devlin *et al.*, "BERT: pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[6] J. Chen *et al.*, "BGE m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," *arXiv:2402.03216*, 2024. [Online]. Available: https://arxiv.org/abs/2402.03216

[7] U. K. Agarwal *et al.*, "Resilience assessment of large language models under transient hardware faults," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2023, pp. 659–670.

[8] K. Ma *et al.*, "Error resilient transformers: A novel soft error vulnerability guided approach to error checking and suppression," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2023.

[9] G. Gavarini *et al.*, "Evaluation and mitigation of faults affecting swin transformers," in *Proc. IEEE Int. Symp. On-Line Test. Robust Syst. Des. (IOLTS)*, Jul. 2023.

[10] L. Roquet *et al.*, "Cross-layer reliability evaluation and efficient hardening of large vision transformers models," in *Proc. Design Autom. Conf. (DAC)*, Jun. 2024.

[11] X. Xue *et al.*, "Soft error reliability analysis of vision transformers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 31, no. 12, pp. 2126–2136, Dec. 2023.

[12] U. Zahid *et al.*, "FAT: Training neural networks for reliable inference under hardware faults," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2020.

[13] F. F. dos Santos *et al.*, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Trans. Reliab.*, vol. 68, no. 2, pp. 663–677, Jun. 2019.

[14] H. Stratigopoulos *et al.*, "Testing and reliability of spiking neural networks: A review of the state-of-the-art," in *Proc. IEEE Int. Symp. Defect Fault Toler. VLSI Nanotechnol. Syst. (DFT)*, Oct. 2023.

[15] W. J. Dally *et al.*, "Evolution of the graphics processing unit (GPU)," *IEEE Micro*, vol. 41, no. 6, pp. 42–51, Nov./Dec. 2021.

[16] M. Véstias and H. Neto, "Trends of CPU, GPU and FPGA for high-performance computing," in *Proc. Int. Conf. Field-Program. Log. Appl. (FPL)*, Sep. 2014.

[17] V. V. Kindratenko *et al.*, "GPU clusters for high-performance computing," in *Proc. Int. Conf. Cluster Comput. Workshops*, Aug./Sep. 2009.

[18] C. Raffel *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, Jan. 2020.

[19] G. Yenduri *et al.*, "Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions," *arXiv:2305.10435*, 2023. [Online]. Available: https://arxiv.org/abs/2305.10435

[20] A. Mahmoud *et al.*, "PyTorchFI: A runtime perturbation tool for DNNs," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun./Jul. 2020, pp. 25–31.

[21] S. Ainsworth and T. M. Jones, "ParaMedic: Heterogeneous parallel error correction," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 201–213.

[22] G. Li *et al.*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2017.

[23] K. S. Yim *et al.*, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2011, pp. 287–300.

[24] F. Su *et al.*, "Testability and dependability of AI hardware: Survey, trends, challenges, and perspectives," *IEEE Des. Test*, vol. 40, no. 2, pp. 8–58, Apr. 2023.

[25] A. Ruospo *et al.*, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, Feb. 2023.

[26] M. H. Ahmadilivani *et al.*, "A systematic literature review on hardware reliability assessment methods for deep neural networks," *ACM Comput. Surv.*, vol. 56, no. 6, Jan. 2024.

[27] P. Rech, "Artificial neural networks for space and safety-critical applications: Reliability issues and potential solutions," *IEEE Trans. Nucl. Sci.*, vol. 71, no. 4, pp. 377–404, Jan. 2024.

[28] Z. Chen *et al.*, "TensorFI: A flexible fault injection framework for tensorflow applications," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2020, pp. 426–435.

[29] S. K. S. Hari *et al.*, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2017, pp. 249–258.

[30] T. Tsai *et al.*, "NVBitFI: Dynamic fault injection for GPUs," in *Proc. 51st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2021, pp. 284–291.

[31] B. Fang *et al.*, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2014, pp. 221–230.

[32] A. Vaswani *et al.*, "Attention is all you need," *arXiv:1706.03762*, 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[33] N. Muennighoff *et al.*, "MTEB: Massive text embedding benchmark," *arXiv:2210.07316*, 2022. [Online]. Available: https://arxiv.org/abs/2210.07316

[34] D. Zhang *et al.*, "Jasper and Stella: distillation of SOTA embedding models," *arXiv:2412.19048*, 2025. [Online]. Available: https://arxiv.org/abs/2412.19048

[35] X. Zhang *et al.*, "mGTE: Generalized long-context text representation and reranking models for multilingual text retrieval," *arXiv:2407.19669*, 2024. [Online]. Available: https://arxiv.org/abs/2407.19669