IRepair: An Intent-Aware Approach to Repair Data-Driven Errors in Large Language Models

SAYEM MOHAMMAD IMTIAZ, Iowa State University, USA
ASTHA SINGH, Iowa State University, USA
FRAOL BATOLE, Tulane University, USA
HRIDESH RAJAN, Tulane University, USA

Not a day goes by without hearing about the impressive feats of large language models (LLMs), and equally, not a day passes without hearing about their challenges. LLMs are notoriously vulnerable to biases in their dataset, leading to issues such as toxicity, harmful responses, and factual inaccuracies. While domain-adaptive training has been employed to mitigate these issues, these techniques often address all model parameters indiscriminately during the repair process, resulting in poor repair quality and reduced model versatility. In this paper, drawing inspiration from fault localization via program slicing, we introduce a novel dynamic slicing-based intent-aware LLM repair strategy, *IRepair*. This approach selectively targets the most error-prone sections of the model for repair. Specifically, we propose dynamically slicing the model's most sensitive layers that require immediate attention, concentrating repair efforts on those areas. This method enables more effective repairs with potentially less impact on the model's overall performance by altering a smaller portion of the model. Furthermore, dynamic selection allows for a more nuanced and precise model repair compared to a fixed selection strategy. We evaluated our technique on three models from the GPT2 and GPT-Neo families, with parameters ranging from 800M to 1.6B, in a toxicity mitigation setup. Our results show that *IRepair* repairs errors 43.6% more effectively while causing 46% less disruption to general performance compared to the closest baseline, *direct preference optimization*. Our empirical analysis also reveals that errors are more concentrated in a smaller section of the model, with the top 20% of layers exhibiting 773% more error density than the remaining 80%. This highlights the need for selective repair. Additionally, we demonstrate that a dynamic selection approach is essential for addressing errors dispersed throughout the model, ensuring a robust and efficient repair.

CCS Concepts: • **Software and its engineering** → *Maintaining software*; • **Computing methodologies** → Natural language generation; *Neural networks.*

Additional Key Words and Phrases: SE4AI, Dynamic Program Slicing, Fault Localization, Program Repair, Large Language Model, Data-driven Error

ACM Reference Format:
Sayem Mohammad Imtiaz, Astha Singh, Fraol Batole, and Hridesh Rajan. 2018. *IRepair*: An Intent-Aware Approach to Repair Data-Driven Errors in Large Language Models. In *Proceedings of The ACM International Conference on the Foundations of Software Engineering (FSE) (FSE '25).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Sayem Mohammad Imtiaz, sayem@iastate.edu, Iowa State University, Ames, Iowa, USA; Astha Singh, Iowa State University, Ames, Iowa, USA, asthas@iastate.edu; Fraol Batole, Tulane University, New Orleans, Louisiana, USA, fbatole@tulane.edu; Hridesh Rajan, Tulane University, New Orleans, Louisiana, USA, hrajan@tulane.edu.

FSE '25, June 23–27, 2025, Trondheim, Norway
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The recent advancement in large language model (LLM) capabilities marks a transformative moment in natural language processing (NLP). Owing to the effectiveness of *transformer* in scaling efficiently to the large corpus, LLMs now excel in tasks such as question-answering, text summarization, and code generation [56]. However, despite their impressive capabilities, LLMs are not without their shortcomings. Akin to traditional software, LLMs can exhibit bugs or generate unintended outputs, manifesting as toxicity, harmful responses, factual errors, or hallucinations [16, 19]. The root cause of these issues often lies in the training data itself [16]. LLMs are typically trained on vast, unfiltered datasets, primarily sourced from the internet, using semi-supervised learning techniques [56]. It is impractical to validate such a vast corpus, eliminating biases, factual inconsistencies, and other issues [16]. As a result, LLMs inadvertently inherit and propagate these issues in their outputs.

Existing approaches to mitigate such issues in LLMs primarily operate in three stages [19, 34, 43]: the pre-training stage [19, 24], alignment stages following pre-training, also known as domain-adaptive training (DAT) methods [11, 20, 25, 37, 43], and runtime methods [22, 32, 50, 51]. Runtime methods, while impactful, often address symptoms rather than the root causes of model errors and can introduce computational overhead, limiting their applicability in low-latency scenarios [8, 15, 19, 43]. Pre-training methods, such as Liu et al.'s attention-sharpening regularizer [24], though effective, are costly due to the need for training from scratch, making them more suitable for new models [15, 43]. DAT methods, on the other hand, directly repair pre-trained models and provide an offline approach to error mitigation, making them particularly useful in real-time applications and situations requiring more invasive repair procedures [15, 43].

DAT has two main paradigms: fine-tuning the model with curated data and preference optimization, such as reinforcement learning from human feedback (RLHF) [33]. However, both approaches update model parameters indiscriminately without considering their relevance to the problem at hand. This can decrease the effectiveness of the repair and increase the likelihood of negatively impacting the model's general performance by altering unrelated parameters. To address this, we introduce *IRepair*, a dynamic slicing-based technique for selectively repairing only the intended part of the model.

Our motivation for targeted LLM repair is inspired by the successful application of the 'fault localization followed by program repair' paradigm in traditional software engineering (SE). This approach has demonstrated effectiveness in producing optimal repairs while preserving the program's original structure as much as possible [27, 31, 47]. For instance, Mechtaev *et al.* employ partial MaxSAT constraint solving and component-based program synthesis to localize bugs and generate repairs, focusing on minimizing alterations to the program's structure [27]. Inspired by these works, we adapt and evaluate this paradigm in the context of LLMs to address data-driven errors. Specifically, we propose a method that first localizes the source of errors within the model and then selectively repairs it. This approach aims to produce optimal repairs while preserving model performance by targeting only the relevant parts of the model and leaving unrelated sections unaffected.

To localize the source of errors within the model, we build upon the concept of relevant program slicing in software engineering [46]. Relevant slicing identifies a subset of program statements that could impact a specified slicing criterion [14, 55]. Similarly, we apply these principles to identify and isolate the parts of the model that are most relevant to the errors being addressed.

Recently, slicing techniques have been adapted for deep learning models, offering advantages such as model protection and simplification [54] and vulnerability mitigation during transfer learning [55]. These approaches use a subset of data as the 'slicing criteria' and analyze the model's activations to identify relevant parts of the model. However, these techniques rely on activation values to determine relevance, which is not directly applicable to the transformer architecture used in LLMs (as discussed in § 3). Additionally, these methods are more akin to static slicing, where the relevant sections of the model are selected after training. This results in a 'fixed selection,' which can be useful for various applications as demonstrated in previous works [54, 55]. In contrast, we hypothesize that a dynamic selection technique, applied during the training process, will enable a more nuanced and precise repair of LLMs through domain-adaptive training with curated data.

Inspired by these works, *IRepair* treats faulty data as a 'slicing criterion' to identify error-prone sections of the model during each training pass. By analyzing the gradients of parameters with respect to the negative log-likelihood (NLL) of the faulty data, we pinpoint the components responsible for the unintended faulty responses. We then repair only the identified area while freezing the rest of the model, subject to a Kullback-Leibler (KL) divergence constraint. This approach allows for focused repair efforts on the most critical sections, minimizing disruption to the existing knowledge stored in most model parameters. Moreover, *IRepair* employs a dynamic slicing approach for repairing LLMs, enabling more nuanced and adaptive model repair compared to existing slicing methods that pre-select a fixed area.

To evaluate the effectiveness of our proposed technique, we conducted a case study focused on mitigating toxicity in LLMs. Given their pre-training on extensive corpora in a semi-supervised manner, LLMs are known to perpetuate biases and toxicity present in the data [50]. To assess the efficacy of our repair approach, we detoxified three models from the GPT-2 and GPT-Neo families, ranging from 800M to 1.6B parameters, using *IRepair*. We compared our results against state-of-the-art baselines, employing the pairwise detoxification dataset developed by Lee *et al.* [20]. Specifically, our baselines include representative techniques from different paradigms within domain-adaptive methods, such as Domain-Adaptive Pretraining (DAPT) [11, 13], DAPT with a regularization term on the pre-training mixture to retain general performance during repair [25], and Direct Preference Optimization (DPO) [37], an RL-based preference optimization technique. Note that this study focuses on evaluating pre-trained language models (PLMs) that have not undergone further alignment. The term 'LLM' used throughout the paper specifically refers to these PLMs. In this context, the general performance of the model refers to the quality of language generation from these models, measured in terms of perplexity metric.

We summarize the key contributions and findings of this paper as follows:

- We propose using sensitivity as a measure of relevance for slicing transformer-based language models, addressing architectural challenges unique to these models.
- Our framework not only facilitates targeted repair of LLMs but also adapts dynamically during training by slicing the model as needed.
- Unlike prior techniques, our method introduces a threshold-free slicing approach, eliminating the need for costly tuning, which can be expensive for large models such as LLMs.
- Our analysis shows that the source of errors can be more pronounced in specific areas of the model than in others, and targeted interventions can deliver more efficient repairs compared to indiscriminate approaches.
- Our approach, *IRepair*, significantly outperforms state-of-the-art techniques, demonstrating greater efficiency in error elimination while preserving model generation quality. Specifically, *IRepair* reduces toxicity by 43.6% more than the closest baseline, *DPO*, while causing 46% less disruption to overall performance.
- We also demonstrate that a dynamic selection approach is essential for addressing errors dispersed throughout the model, ensuring a robust and efficient repair.

2 Background

LLMs have revolutionized NLP and SE tasks due to their ability to capture complex patterns and generate human-like text and code. In this section, we provide an overview of the GPT (Generative Pre-trained Transformer) architecture, which serves as a foundation for many modern LLMs [56], including the models used in our study.

The GPT architecture, introduced by OpenAI [36], is based on the transformer model [42]. It consists of multiple layers of transformer blocks, which are the fundamental units of computation in the model. These blocks are also the primary focus of our slicing technique in *IRepair*. Each block contains two main components:

- **Multi-Head Attention**: This mechanism allows the model to focus on different parts of the input sequence simultaneously. Multi-head attention splits the input into multiple 'heads,' each learning to attend to different input aspects. The attention function is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

  where Q, K, and V are query, key, and value matrices, respectively, obtained from linear projections of the input sequence. d_k is the dimension of the key vectors. The attention output is then passed through a feed-forward network (FFN).
- **Feed-Forward Neural Network (FFN)**: This component processes the output of the attention mechanism. It is a neural network that operates on each position in the input sequence independently. The FFN typically consists of two linear transformations with a non-linear activation function in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

  where W_1, W_2, b_1, and b_2 are learnable parameters.

Each transformer block applies layer normalization and residual connections around these components.

3 Approach

Figure 1 presents a high-level overview of the approach used in *IRepair*, which aims to repair data-driven errors in large language models. The approach consists of two primary components: computing the relevant model slice and then repairing the identified slice selectively. In the first stage, we apply the concept of program slicing to language models to identify the slice that requires repair. In the second stage, we repair the identified slice selectively, focusing on the most error-prone sections of the model while minimizing any impact on its general performance. The following sections will explore the detailed steps involved in *IRepair*.

3.1 Problem Formulation

Let $\pi_\theta : X \longrightarrow Y$ denote a pre-trained language model that maps input texts from X and a set of parameters, θ, to corresponding output texts in Y. Consider a bad demonstration dataset, denoted as $D^E = (X^E, Y^E)$, where the response Y^E to a prompt X^E is undesirable. The goal is to ensure that the model π_θ does not produce responses similar to Y^E when given prompts similar to X^E. Additionally, consider a curated or refined dataset, denoted as $D^R = (X^E, Y^R)$, which demonstrates the desirable response Y^R for those error-evoking prompts X^E. In practice, these refined responses can be obtained through various methods, such as human annotation or conditioning a language model [20, 34, 41, 43]. Once the refined data is acquired, the model π_θ is typically repaired via domain-adaptive training either by directly optimizing the model with the curated dataset [11, 20, 43],

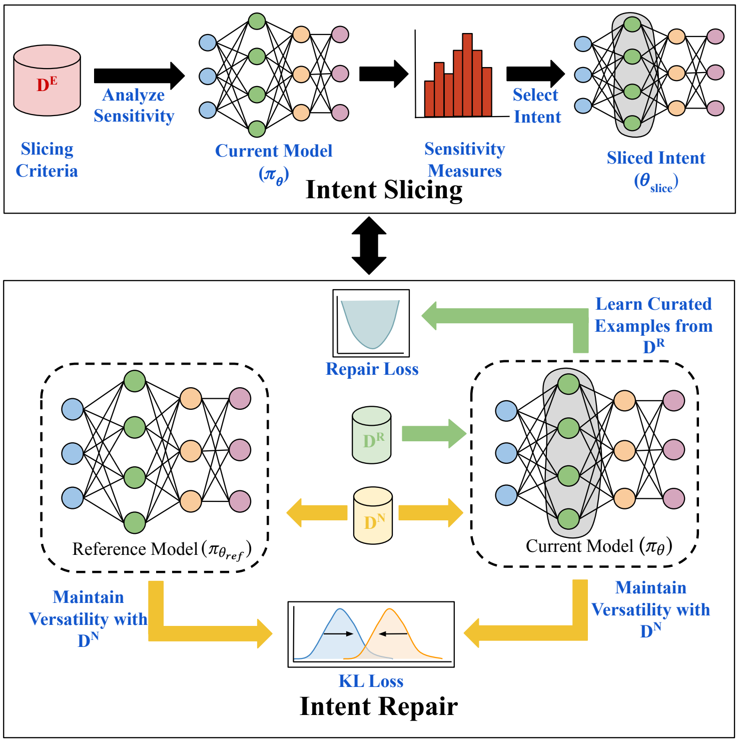

D^E
Slicing Criteria
Analyze Sensitivity
Current Model (π_θ)
Sensitivity Measures
Select Intent
Sliced Intent (θ_{slice})
Intent Slicing
Learn Curated Examples from D^R
Repair Loss
D^R
D^N
Reference Model ($\pi_{\theta_{ref}}$)
Current Model (π_θ)
Maintain Versatility with D^N
Maintain Versatility with D^N
KL Loss
Intent Repair

Fig. 1. Overview of *IRepair*.

or the model is explicitly trained to prefer good demonstrations over bad ones via preference optimization [37].

However, repairing the errors in LLMs often involves a trade-off between overall model performance and repair quality [19, 43]. Higher repair quality tends to come at the cost of reduced general performance [9]. Our experiments also observed this trade-off across various techniques. Domain-adaptive repair methods are particularly susceptible to this issue because they update model parameters indiscriminately without considering their relevance to specific errors. This indiscriminate updating can lead to inefficient error repair, where the reduction in general performance may not justify the extent of error correction achieved.

To address these challenges, we propose focusing repairs on the sections of the model with the highest concentration of errors while leaving unrelated parameters unchanged. This approach aims to enhance repair efficiency, potentially achieving greater repair quality with less disruption to general performance compared to "intent-unaware" or indiscriminate techniques.

To that end, in this paper, we propose using examples from the bad demonstration dataset to identify and slice the most relevant sections of the model, referred to as intent, for selective repair. Specifically, we aim to pinpoint the most error-prone blocks within the transformer architecture for targeted intervention. Additionally, we hypothesize that errors may not be confined to a single block. To accommodate this, we introduce a dynamic slicing mechanism that allows for the selection of the most error-prone sections during the course of training. This approach enables a more nuanced and precise repair of errors throughout the model.

3.2 Slicing Intent

Drawing upon the concept of relevant program slicing [46], in this step, we aim to slice off the most error-prone sections of the model for a selective repair. Such a focused repair approach is critical for LLMs, as updating all parameters using limited repair data may lead to overfitting and knowledge degradation [9]. By selectively slicing the model based on bad data, our aim is to address the root cause of errors in the model while preserving the model's overall knowledge.

3.2.1 Challenges in slicing LLM. Existing slicing techniques for deep learning models are primarily designed for networks using *ReLU* activations [54, 55]. These techniques rely on activation status or their magnitudes to identify relevant parts of the model. However, transformer-based language models employ an attention mechanism with linear transformations, making these methods inapplicable. In contrast to *ReLU* activations, the magnitude of a linear transformation in *attention* doesn't directly correspond to its importance due to subsequent matrix multiplications. As an illustration, consider the following simplified example of unmasked attention scores for a sequence with three tokens, T_1, T_2 and T_3 and an embedding dimension, $d_k = 1$:

$$Q = \begin{bmatrix} 3 \\ -3 \\ 2 \end{bmatrix} \text{ and } K = \begin{bmatrix} -5 \\ 2 \\ 1 \end{bmatrix}$$

$$\text{Score} = \sigma\left(\frac{QK^T}{\sqrt{d_k}}\right) = \sigma\left(\begin{bmatrix} -15 & 6 & 3 \\ 15 & -6 & -3 \\ -10 & 4 & 2 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 0 & 0.95 & 0.05 \\ 1 & 0 & 0 \\ 0 & 0.88 & 0.12 \end{bmatrix} \text{ (simplified)}$$

Here, we observe that in predicting the next token for T_2, T_1 is the most influential. Thus, the multiplication of two negative values of Q and K, -3×-5, yields the highest score for T_2. Unlike *ReLU*, where a node value less than zero might indicate irrelevance [55], in this context, negative values do not reliably signify insignificance. Additionally, considering the magnitude of the activation level is inapplicable [54], as the sign plays a crucial role in the score computation.

In addition, existing techniques require careful calibration of a threshold to select the slice [54, 55], which is a challenging task for large-scale models with billions of parameters, such as LLMs. Tuning for the optimal threshold can be very challenging and time-consuming for such models, highlighting the need for a threshold-free slicing approach for LLMs.

3.2.2 Our approach. To address these challenges, we propose a *gradient-based* approach for determining the relevance of model parameters to the slicing criterion, which does not rely on the activation of neurons. Specifically, we identify the relevant transformer block, referred to as *intent*, of the model by assessing the sensitivity of the blocks to the slicing criterion. The approach works by treating a sample of bad data, ($D^E = (X^E, Y^E)$), as criteria for slicing the intent that requires fixing. Figure 1 shows the overview of our proposed algorithm for slicing the LLM, which involves two major steps: assessing the relevance of parameters to the slicing criteria and intent or slice selection, which will be discussed below:

Computing sensitivity to slicing criteria. As previously discussed, activation-based approaches are not applicable in the context of the *transformer*. Instead, we identify the *intent* by assessing the sensitivity of blocks to *slicing criteria*. To that end, we propose leveraging *negative log-likelihood (NLL)* of the model response to assess the relevance of blocks to the slicing criteria. Specifically, we

Algorithm 1 Slicing Intent

```
1: function NLL($\pi_\theta, X, Y$)
2:     $logits \leftarrow \pi_\theta(X)$                                ▷ Obtain last layer output
3:     $logits \leftarrow logits[:, -1, :]$                             ▷ Omit last token
4:     $P \leftarrow \sigma(logits)$                                    ▷ Perform softmax operation
5:     $mask \leftarrow Y \neq \pi_\theta.padding\_id$
6:     $n_1 \leftarrow -Y \times \log(P)$                               ▷ NLL for tokens
7:     $n_2 \leftarrow \sum_t^T n_1(t) \times mask$                     ▷ NLL for sequence
8:     $nll \leftarrow \frac{n_2}{\sum_t^T mask(t)}$                    ▷ Mean NLL
9:     return $nll$                                                     ▷ Return NLL
10: end function
1: function Sensitivity($\pi_\theta, X, Y$)
2:     $S \leftarrow \{\}$
3:     $L \leftarrow \text{NLL}(\pi_\theta, X, Y)$                      ▷ Get Negative Log Likelihood
4:     for every $block \in \pi_\theta$ do
5:         $S\{block\} \leftarrow \sqrt{\sum_i^{\vartheta \in \theta_{block}} \left(\frac{\partial \mathcal{L}}{\partial w_i}\right)^2}$    ▷ L2-norm
6:     end for
7:     return $S$
8: end function
1: function Slice($\pi_\theta, X, Y$)
2:     $S \leftarrow \text{Sensitivity}(\pi_\theta, X, Y)$              ▷ Measure sensitivity
3:     $B \leftarrow \arg\max_{block \in \pi_\theta} S\{block\}$        ▷ Get most error-prone block
4:     $\theta_{\text{slice}} \leftarrow \{\vartheta | \vartheta \in \theta_B \wedge \theta_B \in \theta\}$    ▷ Get slice
5:     return $\theta_{\text{slice}}$
6: end function
```

first measure the impact of all parameters on the model's response by calculating the first-order gradient of the *NLL* of the generated response. Then, we compute the sensitivity of the block by taking the *L2-norm* of the gradients of all parameters within the block. Without loss of generality, the sensitivity of a block to a slicing criterion, x, can be represented as:

$$S\{\text{block}\} \approx \left\| \nabla_{\theta_{\text{module}}} \left(-\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{1:t-1}) \right) \right\|_2$$

Here, θ_{block} represents the parameters within a specific block of the transformer, $p_\theta(x_t \mid x_{1:t-1})$ represents the probability of the token x_t given the prior tokens $x_{1:t-1}$, T denotes the total number of tokens in x, and θ refers to the overall parameter space. The notation $|| \cdot ||_2$ denotes the L2 norm, which is applied to the gradient of the NLL with respect to the block's parameters.

The *NLL* reflects the model's confidence in generating the target response. A lower NLL score indicates higher confidence in accurately predicting the target response. Taking the gradients of parameters with respect to the *NLL* provides a measure of their sensitivity to the inputs or criteria provided. If a small increase or perturbation in a parameter leads to a noticeable impact on the model output, that parameter is likely important or relevant to the criteria [18]. The greater the magnitude or norm of the gradient for a parameter, the more relevant it is to the criteria.

The *Sensitivity* method (in Algorithm 1) provides our approach for computing sensitivity to slicing criteria. The method takes an instance of the model (π_θ) and slicing criteria as X and Y. It first calculates the *negative log-likelihood* by invoking the *NLL* method in line 3.

To achieve this, the *NLL* method first obtains the model logits (the output of the last layer) by forward passing the input through the model (Line 2). The last generated token is then discarded, as the corresponding token in the ground truth response does not exist (Line 3). Using the *softmax* activation function, the probability distribution for all output tokens in the vocabulary is calculated (Line 4). To compute the *NLL*, a loss mask is obtained where the special padding tokens are skipped to eliminate their impact on the calculation (Line 5). The *NLL* for each token in the sequence is calculated individually by multiplying the probability distribution of ground truth response (Y) with the log-likelihood of the output ($\log(P)$) (Line 6). Next, the *NLL* for the entire sequence is calculated by summing the individual NLLs for every token in the sequence, with padding tokens eliminated by multiplying by the *mask* (Line 7). Finally, in Line 8, the mean *NLL* is computed by dividing by the count of non-padding tokens in the sequence. The final averaged *NLL* approximates the model's confidence in generating the target responses, Y, for the given inputs, X, and returned from the method as the final outcome.

In the *Sensitivity* method, after obtaining the *NLL* for the given criteria, the magnitude of the gradients for each transformer block is calculated by taking the *L2 norms* of all parameters within the block with respect to the *NLL* (Lines 4–5). Specifically, the first-order gradients for all the model parameters are computed with respect to the *NLL* (i.e., $\nabla_\theta(L)$). For each parameter within a transformer block ($\vartheta \in \theta_{\text{block}} \wedge \theta_{\text{block}} \subset \theta$), the gradients are squared, and the square root of their summation is taken to yield the overall magnitude or sensitivity of the block (S) (Line 4). This measure indicates the relevance of the block to the given slicing criteria and is used in the *Slice* method to identify and slice the most error-prone block.

Selecting Intent. The final method, *Slice*, in Algorithm 1, slices the most error-prone block of the model for the provided criteria by leveraging the other two methods. The method takes as input an instance of the model, denoted by π_θ, where θ represents the parameter space of the model, and a set of slicing criteria, X and Y. It first computes the sensitivity of every block in the model by invoking the *Sensitivity* method (Line 2). Then, in Line 3, the block with the highest sensitivity—deemed most relevant to the provided criteria—is selected. When the provided criteria correspond to a sample from poor demonstration data D^E, this block represents where the most error is concentrated. This step effectively eliminates the need for thresholding to identify the slice. Next, in Line 4, the parameters within the selected block are sliced off and returned by the method.

3.3 Repairing Intent

In this step, the identified slice or intent, θ_{slice}, which is primarily responsible for undesirable generation, is addressed through two optimization objectives, as shown in Figure 1. Specifically, our loss function includes an NLL term as repair loss and a KL term to preserve the model's normal utility, as shown in Equation 3.

$$\begin{aligned} Loss =& \alpha \cdot \text{NLL}\left(p_{\theta_{\text{slice}}}(\cdot \mid X^R)\right) \\ &+ KL\left(p_{\theta_{\text{slice}}}(\cdot \mid X^N) \parallel p_{\theta_{\text{ref}}}(\cdot \mid X^N)\right) \end{aligned} \tag{3}$$

Here, α represents the strength of the repair or NLL loss, θ_{slice} denotes the set of sliced parameters from the model π_θ, θ_{ref} represents the parameter space of the reference model $\pi_{\theta_{\text{ref}}}$, and $p_{\theta_{\text{slice}}}$ denotes the probability distribution of π_θ conditioned on the sliced parameters. The key components of the repair process are briefly described below:

Algorithm 2 Repairing Intent

```
1:  function REPAIR($\pi_\theta$, $\pi_{\theta_{ref}}$, $D^E$, $D^R$, $D^N$, $\alpha$)
2:      repeat
3:          $X^R, Y^R \leftarrow \text{batch}(D^R)$                          ▷ Get a good batch
4:          $X^E, Y^E \leftarrow \text{batch}(D^E)$    ▷ Get corresponding bad batch as slicing criteria
5:          $X^N, Y^N \leftarrow \text{batch}(D^N)$                        ▷ Get a normal batch
6:          $\theta_{\text{slice}} \leftarrow \text{Slice}(\pi_\theta, X^E, Y^E)$                ▷ Sliced parameters
7:          $L_1 \leftarrow \text{NLL}(X^R, Y^R, \theta_{\text{slice}})$                     ▷ Repair loss
8:          $L_2 \leftarrow \text{KL}(X^N, Y^N, \theta_{\text{slice}}, \theta_{\text{ref}})$                  ▷ KL loss
9:          $L \leftarrow \alpha \cdot L_1 + L_2$                                 ▷ Total loss
10:         $G \leftarrow \nabla(L, \theta_{\text{slice}})$                      ▷ Gradients w.r.t slice
11:         $update(\theta_{\text{slice}}, G)$                    ▷ Update parameters of $M$
12:     until convergence
13:     return $\pi_\theta$                                      ▷ Return repaired model
14: end function
```

3.3.1 Repair Loss. We use the negative log-likelihood (*NLL*) as the repair loss for our technique. This loss aims to maximize the log-likelihood of the curated responses (Y^R) for fault-evoking prompts (X^E). *NLL* is a commonly employed loss function for repairing models via continued pre-training or supervised fine-tuning [11, 12, 20, 43]. However, unlike existing techniques, we only optimize the sliced parameters (θ_{slice}) of the patient model, π_θ. This selective approach allows for more focused and aggressive updates of the error-prone parameters, potentially leading to more effective repairs. Additionally, updating a smaller portion of the total parameters reduces general performance degradation, as most of the model retains its original parameters. The relative importance of this term is regulated by the α coefficient.

3.3.2 KL Loss. KL loss is used to preserve the general performance of the model during the repair process. Specifically, this term aims to minimize the divergence between a reference distribution (the output of the reference model, $\pi_{\theta_{\text{ref}}}$) and the target distribution (output of π_θ on the pre-training corpus D^N). This term essentially encourages the model to maintain similar generation capabilities to the reference model on unrelated aspects.

3.3.3 Dynamic Slicing. Finally, we employ a dynamic slicing mechanism that selects the most error-prone block of the model during the course of training for an adaptive repair. This design decision is motivated by three key factors:

Error concentration. First, our threshold-free slicing technique selects only the most relevant or error-prone block of the model based on the criteria. However, a single block may not be solely responsible for undesirable responses to certain prompts. Other parts of the model might also significantly contribute to erroneous outputs, as we empirically confirm in our analysis. We find that errors can span multiple blocks, necessitating the repair of more than one block (details in § 4.4). In such cases, repairing only a predetermined fixed block may not be sufficient.

Error movement. Second, a fixed selection strategy, like those used in existing works [54, 55], may fail to adapt to the effects of training dynamics on the model. While an area of the model might appear most responsible for undesirable responses before repair, it may not remain the most error-prone block throughout the course of training. Once training adequately addresses the initially selected area, another unselected area may appear more problematic, deserving more repair effort at that point. A dynamic slicing technique that accounts for the impact of training dynamics can more effectively address such shifts in error concentration.

Local error correction. As demonstrated in our algorithm for repairing intent (Algorithm 2), we use corresponding bad examples for each batch of good examples (Line 3) from the bad demonstration dataset, D^E, as criteria to slice the most relevant block of the model (Line 4). This approach ensures that repair efforts focus on the block that most amplifies errors for the current batch of data. In contrast to pre-selection strategies, which often use all or a sample of data to determine which part to slice [54, 55], our method enables a more nuanced repair by allowing for localized error correction.

3.3.4 Algorithm Overview. The *Repair* method in Algorithm 2 outlines the procedure for repairing the intent using our proposed optimization objectives. The method takes as input the affected or to-be-repaired model π_θ, a reference model $\pi_{\theta_{\text{ref}}}$, which is the same model as initial π_θ and is used to maintain similar performance on unrelated aspects post-correction, and references to bad examples (D^E), good examples (D^R), and normal examples (D^N). Additionally, a hyper-parameter α is provided, which is used as a measure of the strength of the repair loss.

The repair process begins by sampling a batch of good examples in Line 3 during each training iteration. It then constructs a batch of corresponding bad examples to use as slicing criteria for the current iteration (Line 4). Additionally, a random batch from the normal examples is obtained to compute a KL term (Line 5). Next, the *Slice* method is invoked to extract the most error-inducing block of the current model, π_θ (Line 6). The *NLL* loss for the sliced parameters, θ_{slice}, is computed using the good batch in Line 7. Similarly, a KL term is calculated for both the currently repaired model and the reference model using the batch of normal examples (Line 8). In Line 9, the combined loss is obtained, with the *NLL* term regulated by a user-defined coefficient (α). After computing the loss, gradients with respect to the sliced parameters are computed in Line 10 and updated in Line 11. The repair process continues until convergence or early stopping is triggered and the repaired model is returned.

4 Evaluation

In this section, we introduce our evaluation setup, outline our research questions, and discuss the experimental results in detail. As previously mentioned, we evaluate our technique within a model detoxification framework, where our goal is to repair toxic models using the principles outlined in *IRepair*. To this end, we examine our framework across three research questions:

- **RQ1:** *How effectively can IRepair repair or detoxify the model?* This research question evaluates *IRepair*'s effectiveness in eliminating toxicity from models and compares it against several state-of-the-art baseline techniques.
- **RQ2:** *What is the computational overhead of IRepair?* This research question measures the computational overhead of *IRepair* by assessing total floating point operations (FLOPS), peak memory usage, and convergence duration and compares these metrics against baseline techniques.
- **RQ3:** *Does the dynamic selection employed by IRepair offer any advantage?* In this research, we conduct ablation studies and empirical analyses of error concentration in the model to assess the impact and necessity of selective and dynamic repair.

4.1 Experimental setup

4.1.1 Model. We evaluate *IRepair* across three models from the GPT family, namely GPT-2 Large (812M parameters), GPT-2 XL (1.61B parameters), and GPT-Neo (1.3B parameters). The GPT-2 models, developed by *OpenAI*, were trained on 8 million web pages from the *WebText* dataset [36]. The GPT-Neo model, developed by *EleutherAI*, was trained on the *PILE* dataset [1]. We load these pre-trained models from the official Hugging Face repositories of OpenAI and EleutherAI [2, 3].

4.1.2 Dataset. To evaluate *IRepair*, we used the detoxification dataset developed by Lee *et al.* [20], consisting of 24,576 toxic and non-toxic pairs generated by GPT-2 models [36] from prompts sampled from the WikiText-2 train split [29]. The dataset is balanced with an equal number of toxic and non-toxic examples for detoxification. In our experimental setup, we treat the toxic continuations as a bad demonstration dataset, D^E, and the non-toxic continuations as a good dataset, D^R. The evaluation data includes a test set and a small development sample, which consists of 50 challenge prompts from REALTOXICITYPROMPTS [11] and a subset of the WikiText-2 test split, totaling around 32.7K tokens. This development set is used for tuning hyperparameters through repeated runs with various combinations.

Additionally, for each model, we construct an individual normal dataset, D^N, a curated pre-training subset that preserves diversity while minimizing targeted error symptoms. We leverage the notion of 'unconditional generation' to construct this dataset [43]. Specifically, starting with the special start-of-sequence token (GPT models use '<|endoftext|>' as the start-of-sequence token [43]), we generate approximately 15,000 texts for each model using different random seeds. Following prior work, we employ nucleus sampling with a temperature of 1 and $p = 0.9$ during generation [43]. The unconditionally generated text corpus is considered a good representative of the model's training corpus [43]. Then, we score each generation using perspective API and remove the ones with toxicity scores higher than 0.5 [11], removing approximately 1.2% of the initial dataset. By minimizing the KL divergence in these examples, we aim to preserve the model's ability to generate random text similarly to its pre-repair state, thereby reducing the impact on its general performance.

4.1.3 Baseline. We compare the performance of two variants of *IRepair*: the standard *IRepair*, which does not enforce a KL constraint, and *IRepair + KL*, which does, against several representative state-of-the-art baselines within domain-adaptive training, as introduced below:

Domain-Adaptive Pretraining (DAPT). DAPT is a framework introduced by Gururangan *et al.* [13], which involves continuing the pretraining of a model on domain-specific texts. Gehman *et al.* [11] applied the DAPT framework to further train GPT-2 models on nontoxic texts to detoxify them. In our setup, we evaluate the effectiveness of DAPT in detoxifying models and compare it with *IRepair*.

Direct Preference Optimization (DPO). DPO is a cutting-edge algorithm designed to replace RLHF (Reinforcement Learning from Human Feedback [33]) due to its complex and unstable training process. It directly steers the model towards desirable generations over undesirable ones [37]. The DPO algorithm has been shown to effectively eliminate toxicity from models, as demonstrated by Lee *et al.* [20]. We also compare our method against DPO.

Domain-Adaptive Pretraining with KL Constraint (DAPT+KL). We also compare *IRepair* against a variant of the DAPT method that includes a KL constraint to preserve the general model performance [25]. While DAPT alone may cause the model to deviate when training on a domain-specific corpus, adding a KL term helps evaluate the repair quality of this approach by ensuring that the model maintains its general performance while focusing on domain-specific adjustments.

Additionally, in RQ3, we compare *IRepair* against two of its variants to assess the effectiveness of the components employed by *IRepair*, as described below:

IRepair (Min). In this variant of *IRepair*, during each training iteration, instead of selecting the block with the highest error concentration, the block with the least concentration is chosen. Comparing *IRepair* against this variant allows us to assess the impact of selective repair.

IRepair (Fixed). In this variant of *IRepair*, a fixed slice of the model is pre-selected for repair. Specifically, using a substantial random sample of bad data (D^E) consisting of 2000 examples, average sensitivities for all parameters are computed. Then, using the same technique described in Algorithm 1, block sensitivity is computed, and the block with the highest sensitivity is selected for repair. Comparing *IRepair* against this variant allows us to assess the impact of dynamic selection.

4.1.4 Metric. Following the prior works [11, 12, 20], we use the following two metrics to evaluate the toxicity and general performance of the model after detoxification.

Toxicity. Gehman *et al.* developed a dataset called *REALTOXICITYPROMPTS* to evaluate toxicity in LLMs [11]. This dataset consists of sentence-level prompts that are provided to LLMs to generate continuations, which are likely to elicit toxic responses from the models. They also created a *challenge* subset of this dataset, which includes 1,199 prompts that consistently caused all models in their experiments to generate toxic responses [11]. This *challenge* subset has been used in previous studies to evaluate the effectiveness of detoxification methods [12, 20]. Similarly, we leverage this subset to assess the detoxification quality of our repaired models. Additionally, following prior works [11, 12, 20], we use the widely adopted toxicity detection tool, *PERSPECTIVE API* [5], to assign a toxicity score to each generation. The score ranges from 0 to 1, with higher scores indicating more toxic responses. The toxicity score for the test data is calculated as the average toxicity score across the entire test set, following prior works [20].

Perplexity. Perplexity is a widely used metric to evaluate the generation quality of language models. It has been employed to assess degradation in model generation after the repair process [11, 12, 20, 43]. *Perplexity* measures how uncertain or "perplexed" the model is in predicting the next word. Higher perplexity indicates that the model is worse at predicting the next word, meaning its generation quality is lower. When evaluated on a test corpus, it reflects how well the model's generated text aligns with the test data. Following prior works on GPT models [12, 20], we compute the *perplexity* of the model before and after repair using the test split of *Wikitext-2* [29], which contains 4,358 rows with approximately 241K words. Specifically, To calculate it, we split the test corpus into 1024-token segments, compute the NLL for target tokens in each segment, and weigh the NLL values by the number of target tokens. The perplexity is then computed as the exponentiation of the mean NLL per token.

Additionally, we evaluate the computational overhead of the techniques using the following three metrics, as described in the literature [17, 21, 39]:

TFLOPs. TFLOPs (Tera Floating-Point Operations per Second) represents the total number of floating-point operations performed in the trillions during the course of training or inference [17]. This metric is commonly used to gauge the computational demand of a method. To compute the *TFLOPs*, we utilize an open-source tool [7] that applies the formula derived by Kaplan *et al.* for GPT models [17].

Peak Memory Usage. Memory consumption is a critical factor when developing techniques for large language models (LLMs)[39]. To evaluate this, we measure the peak memory usage of all techniques during training. We employ the method used by Lee *et al.* [21], where an independent process queries the GPU using the *nvidia-smi* command at 1-second intervals to record the highest memory usage observed.

Model	Metric	Vanilla	DAPT	DAPT+KL	DPO	*IRepair*	*IRepair + KL*
GPT2 812M	Toxicity	41.53	11.99	13.93	14.97	$\underline{7.74}$	**5.11**
	Perplexity	19.44	26.95	22.96	24.10	**20.93**	$\underline{22.35}$
GPT2 1.61B	Toxicity	48.22	39.74	$\underline{5.21}$	22.71	10.67	**4.68**
	Perplexity	17.40	22.96	20.13	21.02	$\underline{18.16}$	**18.15**
GPT Neo 1.3B	Toxicity	40.89	31.57	37.93	12.26	$\underline{5.69}$	**4.94**
	Perplexity	14.56	17.00	**16.17**	16.70	16.56	$\underline{16.52}$
Overall	Toxicity	43.6±2.3	27.8±8.2	19±9.8	16.7±3.1	$\underline{8\pm1.4}$	**4.9±0.1**
	Perplexity	17.1±1.4	22.3±2.9	19.8±2.0	20.6±2.1	**18.6±1.3**	$\underline{19\pm1.7}$

Table 1. Comparative Overview of *IRepair*'s Performance. (Toxicity scores are scaled from 0 to 100. The best performance is highlighted in bold, and the second-best is underlined for each model.

GPU Time. We report the total GPU time required for training each technique until convergence. This measurement is obtained using *PyTorch's CUDA* API [6], which tracks the time spent on the GPU throughout the training process.

Total Iteration. Additionally, we report the total iteration needed until the model converges or early stopping is triggered.

4.1.5 Training Details. As discussed in § 4.1.1, we used pre-trained models from the official *Hugging Face* repositories of *OpenAI* and *EleutherAI* [2, 3]. We leveraged Python's deep learning library *PyTorch* [35] for further training these models across all techniques. All hyperparameters in our study were fine-tuned using a small development dataset produced by Lee *et al.* [20]. For *DPO*, *DAPT*, and *DAPT + KL*, we used the implementation provided by Lee *et al.* [20] as a reference.

Similarly, we fine-tuned the hyperparameters for all techniques using the development set. Specifically, the final tuned learning rates for standard *IRepair*, *IRepair + KL*, *DPO*, *DAPT*, and *DAPT+KL* are $2e^{-5}$, $5e^{-5}$, $1e^{-6}$, $1e^{-6}$, and $5e^{-6}$, respectively. Through trial and error, we found that a higher learning rate tends to achieve better repair quality at the expense of general performance and vice versa. Since *IRepair* only modifies a small portion of the model, it can accommodate a larger learning rate with less adverse impact on general performance compared to other indiscriminate techniques. Similarly, *DAPT+KL* allows a slightly higher learning rate than *DPO* and *DAPT* as it explicitly aims to maintain general performance during repair. We also set the value of α to 0.5 for both *IRepair* and *DAPT+KL* after tuning.

For training models using all techniques, we used the memory-efficient *RMSProp* optimizer with 150 warmup steps and a linear learning rate scheduler. A batch size of four was used for the techniques, with a validation split of eight batches, each with a batch size of eight. Models were trained with a validation loss patience of 30 iterations. All models were trained on an *NVIDIA A100* GPU with 40GB of memory. We conducted all the training using the same random seed to ensure reproducibility and enable a fairer comparison.

4.2 RQ1: How effectively can *IRepair* repair the model?

In this research question, we evaluate the repair effectiveness of *IRepair* and compare it against several baselines. Table 1 provides a comparative overview of *IRepair*'s performance across all models. The results show that both variants of *IRepair* consistently outperform all other techniques on every model tested, achieving a higher repair score with better general performance stability. Specifically, standard *IRepair* achieves an average 81.6% reduction in toxicity with an 8.3% increase in perplexity across all models. The *IRepair + KL* variant reduces toxicity by 88.7% with an 11% increase in perplexity.

In contrast, *DPO*, *DAPT+KL*, and *DAPT* reduce toxicity by 61.8%, 56.3%, and 36.2%, while increasing perplexity by 20.3%, 15.3%, and 30.2%, respectively. Thus, both *IRepair* variants clearly outperform all baseline techniques in both metrics. For example, compared to *DPO*, standard *IRepair* and *IRepair + KL* are 32% and 43.6% more effective in reducing toxicity while incurring 59.2% and 46% less increase in perplexity. Similarly, against *DAPT+KL*, standard *IRepair* and *IRepair + KL* are 44.8% and 57.5% more effective in reducing toxicity while showing 45.8% and 28.2% less increase in perplexity. Additionally, we find that all techniques, including *IRepair*, significantly outperform *DAPT*.

Understandably, the ability of *DAPT* to repair the model is limited by its tendency to lose general performance more rapidly. Overall, its perplexity increases by 30.2%, compared to increases of 15.3%, 20.3%, 8.3%, and 11% for *DAPT+KL*, *DPO*, *IRepair*,and *IRepair + KL*, respectively. This clearly shows that adding a KL term to the *DAPT* loss for self-generated random data helps better preserve unrelated model knowledge. Furthermore, the results demonstrate that compared to *DAPT+KL*, which operates on all parameters indiscriminately, *IRepair*'s selective approach is more effective at controlling performance degradation (with 45.8% and 28.2% less perplexity increase than *DAPT+KL*), despite using a higher learning rate. This can be attributed to the fact that *IRepair* only adjusts a fraction of the parameters, leaving most untouched during each training pass, which better preserves overall model performance.

The results also demonstrate that KL-enabled techniques achieve better repair or toxicity scores, as they allow for more aggressive model repair at higher learning rates. For instance, *DAPT+KL* reduces the toxicity score by 55% compared to its non-KL counterpart, *DAPT*. Similarly, *IRepair + KL* achieves a 9% greater reduction in toxicity and exhibits 93% lower standard error, indicating greater stability compared to standard *IRepair*. Even with KL-enabled *DAPT*, both *IRepair* variants significantly outperform it by 44.8% and 57.5% in toxicity reduction while also resulting in 59.2% and 46% less increase in perplexity. *IRepair*'s ability to support higher learning rates is a crucial factor in its effectiveness. However, this added efficiency is also largely driven by *IRepair*'s focused repair approach, as demonstrated in § 4.4. *IRepair*'s selective strategy not only enables aggressive model repair with minimal loss in general performance but also targets the most relevant or error-prone sections of the model, leading to the observed high repair efficiency.

4.3 RQ2: What is the computational overhead of *IRepair*?

In this research question, we evaluate the computational overhead of *IRepair* and compare it against baseline techniques. Table 2 presents the overhead of various techniques across four metrics. Among the two *IRepair* variants, the overhead of standard *IRepair* is more amenable to the other three baseline techniques across all metrics. For example, it ranks first in memory consumption and second in GPU time, despite incurring higher TFLOPs and requiring more iterations to converge.

The additional compute units (TFLOPs) consumed by *IRepair* variants are due to the extra forward pass and a higher number of iterations required for convergence. *IRepair + KL* involves four forward passes: one for a toxic batch of data to assess the sensitivity (D^E), one for a non-toxic batch (D^R), and for normal data, one pass through the model under repair (π_θ) and another through the reference model ($\pi_{\theta_{\text{ref}}}$). In contrast, standard *IRepair* does not compute the KL term, thereby eliminating two forward passes for normal data, which results in significantly lower TFLOPs overall—53% less than *IRepair + KL*. *DPO* also requires four forward passes; however, it converges in fewer iterations, leading to lower total TFLOPs.

On a per-token basis, the TFLOPs required for *IRepair* are comparable to other baselines (with standard *IRepair* ranking second and *IRepair + KL* requiring the most, though comparable to *DPO*). However, due to the higher number of iterations needed for convergence to address a smaller part of the model, total TFLOPs are higher. Despite this, *IRepair* 's GPU time remains proportionally lower, and it trains faster or comparably to some baseline techniques, such as *DAPT* and *DAPT+KL*.

Model	Metric	DAPT	DAPT+KL	DPO	IRepair	IRepair + KL
GPT2 812M	TFLOPs/Token	4.92	8.19	9.83	6.89	10.17
	TFLOPs	12.32	24.82	18.47	45.97	63.19
	GPU Time (sec)	1,994	1,411	733	2032	1,933
	Peak Memory Usage (MiB)	13175	16547	18677	13990	17071
	Total Iteration	6200	3750	2325	8250	5125
GPT2 1.61B	TFLOPs/Token	9.80	16.33	19.60	13.52	20.05
	TFLOPs	46.63	65.34	35.24	51.34	121.53
	GPU Time (sec)	7,398	3,438	1,304	2010	2,769
	Peak Memory Usage (MiB)	27664	32411	35659	25042	29333
	Total Iteration	11775	4950	2225	4700	5000
GPT Neo 1.3B	TFLOPs/Token	8.47	14.12	16.94	11.92	17.57
	TFLOPs	10.26	19.39	27.37	19.03	64.41
	GPU Time (sec)	1,416	747	1,013	545	1,193
	Peak Memory Usage	22842	26233	25853	20356	23179
	Total Iteration	3000	1700	2000	1975	3025
Overall	TFLOPs/Token	**7.73**	12.88	15.46	10.78	15.93
	TFLOPs	**23.07**	36.52	27.03	38.78	83.04
	GPU Time (sec)	3,603	1,865	**1,017**	1529	1,965
	Peak Memory Usage (MiB)	21,227	25,064	26,730	**19796**	23,194
	Total Iteration	6,992	3,467	**2,183**	4975	4,383

Table 2. The Computational Overhead of the *IRepair* compared with Baseline Techniques.

This could be attributed to better GPU utilization in *IRepair* variants, which process more TFLOPs per iteration, while *DAPT* takes longer to converge, spreading out its TFLOPs and leading to lower overall GPU utilization.

In terms of memory consumption, we find that standard *IRepair* uses the least memory, while *IRepair + KL* ranks third. This is because the backward pass in *IRepair* is more constrained than in other techniques. First, to calculate sensitivity, it only computes gradients for the transformer blocks, excluding the *embedding* and final output layers. After slicing the layer, it zeroes out the gradients, freeing memory. In the second backward pass, it only computes gradients for the required smaller slice, resulting in lower peak memory consumption compared to other techniques. *IRepair + KL* requires slightly more memory than *DAPT* due to storing extra logits for normal data from two forward passes.

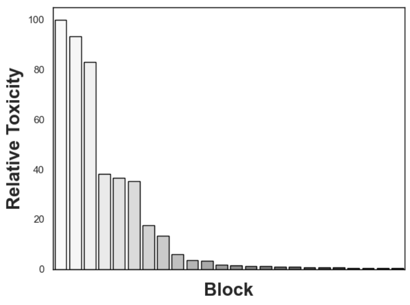
Overall, while *IRepair* incurs higher TFLOPs due to longer iterations, it remains memory-efficient and trains reasonably faster by fully utilizing available GPU power. In exchange for additional compute units, *IRepair* offers better repair efficiency than the other techniques. Between *IRepair* and *IRepair + KL*, although the latter is more computationally intensive, it provides greater stability in model repair, as observed in § 4.2.

4.4 RQ3: Does the dynamic selection employed by *IRepair* offer any advantage?

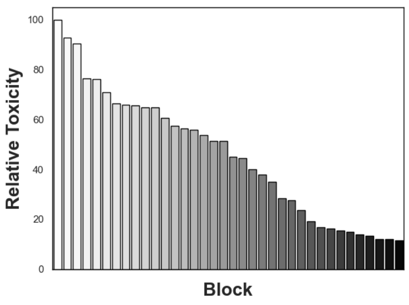
In the final research question, we investigate the effectiveness of dynamic slicing in delivering focused model repair. As described in § 4.1.3, we evaluated two additional variants of both the standard *IRepair* and *IRepair + KL*: *IRepair + Min* and *IRepair + Fixed*. The *IRepair + Min* variant selects the transformer block with the lowest error concentration, as opposed to the highest in the regular *IRepair*. This baseline allows us to assess the impact of selection on repair efficacy. Similarly, *IRepair + Fixed* disables dynamic slicing and instead pre-selects the block with the highest error concentration for repair. This variant enables us to assess the impact of dynamic selection on repair effectiveness.

Model	Metric	IRepair			IRepair + KL		
		Fixed	Min	Max	Fixed	Min	Max
GPT2 812M	Toxicity	45.53	36.84	7.74	39.04	42.08	5.11
	Perplexity	20.78	21.59	20.93	22.88	24.77	22.35
GPT2 1.61B	Toxicity	37.36	47.63	10.67	4.32	47.25	4.68
	Perplexity	18.50	20.95	18.16	18.80	18.01	18.15
GPT Neo 1.3B	Toxicity	39.71	38.79	5.69	40.57	40.86	4.94
	Perplexity	14.85	14.98	16.56	14.86	15.01	16.52
Overall	Toxicity	40.87	41.08	8.03	27.98	43.40	4.91
	Perplexity	18.04	19.18	18.55	18.85	19.26	19.01

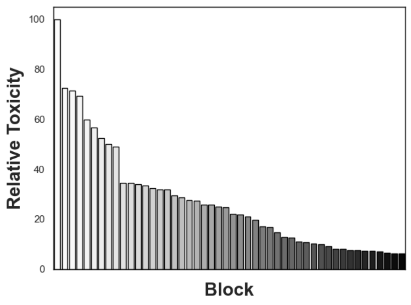
Table 3. Impact of Dynamic Slicing on Repair Efficacy



(a) GPT Neo 1.3B



(b) GPT2 812M



(c) GPT2 1.61B

Fig. 2. Relative Toxicity Levels of Transformer Blocks Across Different Models

Table 3 presents the comparative results of these variants against the regular *IRepair*. It demonstrates that both regular *IRepair* variants significantly outperform their *Min* and *Fixed* counterparts. For instance, standard *IRepair* and *IRepair + KL* reduce toxicity by 80.5% and 88.6% more than their *Min* variants while maintaining a similar level of perplexity (with regular *IRepairs* showing 3.3% and 1.3% more reduction, respectively). It clearly shows the impact of selection made by regular *IRepair* in repairing the model.

Similarly, both regular *IRepairs* outperform their *Fixed* counterparts by a clear margin. Standard *IRepair* and *IRepair + KL* reduce toxicity by 80.4% and 82.5% more than their *Fixed* counterparts while achieving slightly lower perplexity (with *Fixed* variants scoring 2.7% and 0.8% less in perplexity).

Additionally, pre-selecting the most error-prone blocks and focusing repair efforts on them slightly outperforms the dynamic *Min* variants, with the difference being more noticeable in the *Fixed + KL* variant (scoring 35% less in toxicity than *Min*).

We particularly observed that the *Fixed + KL* approach performs comparably to the *IRepair + KL* method on the GPT2 1.61B model. To understand why fixed selection was effective for this model, we analyzed the toxicity levels of all transformer blocks across different models. We calculated toxicity by randomly sampling 2000 examples and computing the average sensitivity for each block. Figure 2 displays the distribution of relative toxicity across all blocks for the models studied.

For the GPT2 1.61B model, the most toxic block is about 28% more toxic than the second most toxic block. In contrast, for the GPT2 812M model, the difference between the most and second most toxic blocks is only 8% and 9.5% with the third most toxic blocks. Similarly, for the GPT Neo 1.3B model, the difference between the most and second most toxic blocks is just 6.5%. This indicates that the GPT2 1.61B model has a higher concentration of toxicity in the top block, making fixed targeted repair of this block more effective. In other models, errors are more evenly distributed among the top few blocks, reducing the effectiveness of a top-block-only repair.

This suggests that errors can be dispersed throughout the model, and a fixed selection technique may require costly tuning to determine the optimal selection threshold. In contrast, our dynamic slicing approach avoids this need for tuning and allows for model-wide repair by dynamically focusing on the most error-prone areas during training.

Figure 2 also shows that GPT-2 1.61B, GPT-2 812M, and GPT-Neo 1.3B have 245.3%, 120.8%, and 1137.7% higher average error density in the top 20% of blocks compared to the remaining 80% of blocks. Error density was measured by dividing the total toxicity within N blocks by N. This clearly indicates that the repair process should give more priority to these highly error-inducing regions than to others, which may lead to superior outcomes, as observed in our results.

5 Related work

Software engineering research has proposed various techniques to repair bugs in deep neural networks (DNNs) that arise during training or within the network structure itself [26, 45, 52, 53]. Examples include Zhang *et al.* 's method for monitoring DNN training and suggesting corrective actions for anomalies [53], and Wardat *et al.* 's work on identifying and fixing structural bugs in DNNs [45]. However, these techniques primarily address issues stemming from the DNN itself. In contrast, data-driven errors in LLMs, such as toxicity or hallucinations, can stem from biases and inconsistencies within the training data itself [16], requiring solutions beyond structural or training bug fixes.

On the other hand, machine learning research offers several strategies to mitigate such errors, broadly categorized into three approaches [19, 34, 43]: inference or decoding-time methods [8, 22, 23, 32, 38, 43, 48, 50, 51], pretraining-based methods [19], and domain-adaptive training methods [11, 20, 25, 37, 43]. Decoding-time methods aim to circumvent problematic responses during inference by techniques such as vocabulary shifting, word banning, or response filtering [8, 43]. However, they do not address the root causes of errors within the model and often fail to consider the sequence-level semantics of generated text [43]. There are also prompt-based techniques that resemble decoding-time methods, relying on prompt engineering to avoid undesirable responses [10, 49]. However, these techniques do not address the underlying errors within the model and are particularly suited for dialogue systems.

In contrast to decoding-time methods, pretraining-based approaches suggest removing problematic data from the training corpus. While effective, this can be prohibitively expensive [43].

The domain-adaptive model adjustment, on the other hand, has emerged as a promising strategy, offering a balance between the simplicity of decoding-time methods and the effectiveness of pretraining-based approaches [20, 43].

Domain-adaptive methods aim to continue pretraining or fine-tuning models on domain-specific texts [11, 20]. For instance, Gehman *et al.* [11] applied a framework called Domain-Adaptive Pretraining (DAPT) [13] to further pretrain GPT-2 on curated non-toxic data, reducing its toxicity. Similarly, Wang *et al.* [43] used the DAPT framework with self-generated data to detoxify models. However, their technique is model-dependent and relies on self-generated data, which may not be applicable in all error scenarios, such as factual inaccuracies or hallucinations. Solaiman and Dennison [41] proposed a general framework for aligning language models to specific target values, but it involves expensive iterative training [40]. These methods aim to optimize pre-trained model parameters using domain-specific texts, mitigating errors while minimizing the impact on overall performance.

Reinforcement learning (RL) from human feedback (RLHF) is another domain-adaptive method that relies on human demonstration datasets to align language models, and it has been shown to mitigate errors in LMs [33]. The recently proposed Direct Preference Optimization (DPO) is a cutting-edge RL-inspired algorithm designed to overcome the instability and complexity of RLHF [37]. In recent work, Lee *et al.* [20] demonstrated the effectiveness of the DPO framework in detoxifying the GPT-2 model using paired datasets of toxic and non-toxic examples.

However, these existing domain-adaptive techniques treat all model parameters uniformly during repair. This indiscriminate approach increases the risk of altering parameters unrelated to the specific errors being addressed, which can disrupt the general knowledge stored in those parameters. Such an intent-unaware" approach not only risks harming overall performance but is also limited in effectively targeting the error-prone parts of the model. A more focused strategy could address these issues more efficiently by concentrating the repair effort where it is most needed. Our proposed technique, *IRepair*, addresses these concerns by enabling a selective repair strategy.

Knowledge editing (KE) is a related area within machine learning that focuses on updating a model's factual knowledge, allowing developers and end-users to modify the model beyond the training setup [28, 30, 44]. These techniques complement training-time methods by enabling model fixes during test time [30]. While training-time methods aim for global correction, KE techniques provide localized fixes by updating the model's knowledge with a single instance [28, 30, 44].

6 Threats To Validity and Limitations

An internal threat to the study is the quality of the detoxification and evaluation datasets. To address this, we utilize both the training dataset and evaluation setup from a recent reputable work on LLM detoxification [20]. Additionally, we employ the implementations provided in the same study to address concerns about the construct validity of baseline techniques. Another internal threat arises from the reliability of the evaluation metrics. To mitigate this concern, we measure repair or toxicity scores using the widely used Perspective API [5] and evaluate model quality post-repair using perplexity, as employed in many prior works [11, 20, 43]. Similarly, our evaluation metrics for measuring computational overheads are based on well-established metrics in the literature [17, 21]. An external threat is the relevance of the models used. To address this, we have selected three models from the GPT and GPT-Neo families with billions of parameters, all of which have previously been employed for evaluating detoxification techniques [11, 19, 20, 22, 50, 51].

While the case study performed in this paper shows that *IRepair* can effectively address the data-driven errors in large language models (LLMs), it is demonstrated within the context of detoxification. Further research is encouraged to explore its effectiveness and generalizability to other data-driven error scenarios, which will enhance the understanding and potential applications of this approach.

Furthermore, although we evaluated *IRepair* on models with billions of parameters—similar to those frequently used in evaluating prior repair techniques—its performance in ultra-large-scale LLMs remains an area for further investigation.

7 Conclusion

In this paper, we introduce *IRepair*, an intent-aware technique for selectively repairing data-driven errors in LLMs through dynamic model slicing. While domain-adaptive training with curated data has shown promise, it tends to optimize model parameters indiscriminately, which can limit repair efficacy and increase the risk of negatively affecting general model performance by altering unrelated parameters. To address these limitations, *IRepair* identifies the relevant portions of the model responsible for the errors, allowing for more targeted repair and making it an intent-aware approach. Our method employs a gradient-based technique to select the most relevant parts of the model by analyzing sensitivity to slicing criteria. Unlike existing slicing routines, our technique is specifically designed to address transformer-related challenges and to avoid the need for expensive tuning of selection thresholds. In a case study focused on model detoxification, *IRepair* demonstrated its effectiveness in addressing the root causes of toxicity while minimizing the impact on general performance, outperforming state-of-the-art baselines. Our empirical results also suggest that errors can be highly concentrated in very limited regions of the model, highlighting the need for selective repair. We further demonstrate that a dynamic selection-based repair strategy is essential for effectively addressing errors dispersed throughout the model.

8 Data Availability Statement

The replication package is available here [4] and includes all the results, code, and data, along with a 'readme' file that provides detailed instructions on how to reproduce the results.

References

[1] 2024. *GPT Neo Models by Eleuther AI.* Retrieved August 31, 2024 from https://www.eleuther.ai/artifacts/gpt-neo
[2] 2024. *GPT Neo Models by Eleuther AI: HuggingFace Repository.* Retrieved August 31, 2024 from https://huggingface.co/EleutherAI/
[3] 2024. *GPT2 Models by OpenAI: HuggingFace Repository.* Retrieved August 31, 2024 from https://huggingface.co/openai
[4] 2024. *IRepair - results and replication package.* Retrieved August 31, 2024 from https://huggingface.co/datasets/Anonymous007/IRepair/tree/main
[5] 2024. *Perspective API.* Retrieved August 31, 2024 from https://perspectiveapi.com/
[6] 2024. *PyTorch CUDA API.* Retrieved August 31, 2024 from https://pytorch.org/docs/stable/cuda.html
[7] Adam Casson. 2023. Transformer FLOPs. (2023). https://adamcasson.com/posts/transformer-flops
[8] Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. 2019. Plug and play language models: A simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164* (2019).
[9] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2022. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904* (2022).
[10] Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I Liao, Kamilė Lukošiūtė, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, et al. 2023. The capacity for moral self-correction in large language models. *arXiv preprint arXiv:2302.07459* (2023).
[11] Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A Smith. 2020. Realtoxicityprompts: Evaluating neural toxic degeneration in language models. *arXiv preprint arXiv:2009.11462* (2020).
[12] Mor Geva, Avi Caciularu, Kevin Ro Wang, and Yoav Goldberg. 2022. Transformer feed-forward layers build predictions by promoting concepts in the vocabulary space. *arXiv preprint arXiv:2203.14680* (2022).
[13] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964* (2020).
[14] Tibor Gyimóthy, Arpád Beszédes, and Istán Forgács. 1999. An efficient relevant slicing method for debugging. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 303–321.

[15] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* (2023).

[16] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[18] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[19] Tomasz Korbak, Kejian Shi, Angelica Chen, Rasika Vinayak Bhalerao, Christopher Buckley, Jason Phang, Samuel R Bowman, and Ethan Perez. 2023. Pretraining language models with human preferences. In *International Conference on Machine Learning*. PMLR, 17506–17533.

[20] Andrew Lee, Xiaoyan Bai, Itamar Pres, Martin Wattenberg, Jonathan K Kummerfeld, and Rada Mihalcea. 2024. A mechanistic understanding of alignment algorithms: A case study on dpo and toxicity. *arXiv preprint arXiv:2401.01967* (2024).

[21] Youngwan Lee, Joong-won Hwang, Sangrok Lee, Yuseok Bae, and Jongyoul Park. 2019. An energy and GPU-computation efficient backbone network for real-time object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. 0–0.

[22] Chak Tou Leong, Yi Cheng, Jiashuo Wang, Jian Wang, and Wenjie Li. 2023. Self-detoxifying language models via toxification reversal. *arXiv preprint arXiv:2310.09573* (2023).

[23] Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A Smith, and Yejin Choi. 2021. DExperts: Decoding-time controlled text generation with experts and anti-experts. *arXiv preprint arXiv:2105.03023* (2021).

[24] Bingbin Liu, Jordan Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. 2024. Exposing attention glitches with flip-flop language modeling. *Advances in Neural Information Processing Systems* 36 (2024).

[25] Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. 2023. Chain of hindsight aligns language models with feedback. *arXiv preprint arXiv:2302.02676* (2023).

[26] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.

[27] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[28] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems* 35 (2022), 17359–17372.

[29] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[30] Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D Manning. 2021. Fast model editing at scale. *arXiv preprint arXiv:2110.11309* (2021).

[31] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[32] Tong Niu, Caiming Xiong, Semih Yavuz, and Yingbo Zhou. 2024. Parameter-Efficient Detoxification with Contrastive Decoding. *arXiv preprint arXiv:2401.06947* (2024).

[33] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[34] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188* (2023).

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[37] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2024).

[38] Timo Schick, Sahana Udupa, and Hinrich Schütze. 2021. Self-diagnosis and self-debiasing: A proposal for reducing corpus-based bias in nlp. *Transactions of the Association for Computational Linguistics* 9 (2021), 1408–1424.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[40] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Boyd-Graber, and Lijuan Wang. 2022. Prompting gpt-3 to be reliable. *arXiv preprint arXiv:2210.09150* (2022).

[41] Irene Solaiman and Christy Dennison. 2021. Process for adapting language models to society (palms) with values-targeted datasets. *Advances in Neural Information Processing Systems* 34 (2021), 5861–5873.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[43] Boxin Wang, Wei Ping, Chaowei Xiao, Peng Xu, Mostofa Patwary, Mohammad Shoeybi, Bo Li, Anima Anandkumar, and Bryan Catanzaro. 2022. Exploring the limits of domain-adaptive training for detoxifying large-scale language models. *Advances in Neural Information Processing Systems* 35 (2022), 35811–35824.

[44] Mengru Wang, Ningyu Zhang, Ziwen Xu, Zekun Xi, Shumin Deng, Yunzhi Yao, Qishen Zhang, Linyi Yang, Jindong Wang, and Huajun Chen. 2024. Detoxifying Large Language Models via Knowledge Editing. *arXiv preprint arXiv:2403.14472* (2024).

[45] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the 44th international conference on software engineering*. 561–572.

[46] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[47] Ming Wen, Jun jie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*. 1–11.

[48] Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. 2023. Large language models are better reasoners with self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 2550–2575.

[49] Yueqi Xie, Jingwei Yi, Jiawei Shao, Justin Curl, Lingjuan Lyu, Qifeng Chen, Xing Xie, and Fangzhao Wu. 2023. Defending chatgpt against jailbreak attack via self-reminders. *Nature Machine Intelligence* 5, 12 (2023), 1486–1496.

[50] Canwen Xu, Zexue He, Zhankui He, and Julian McAuley. 2022. Leashing the inner demons: Self-detoxification for language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11530–11537.

[51] Zonghan Yang, Xiaoyuan Yi, Peng Li, Yang Liu, and Xing Xie. 2022. Unified detoxifying and debiasing in language generation via inference-time adaptive optimization. *arXiv preprint arXiv:2210.04492* (2022).

[52] Hao Zhang and WK Chan. 2019. Apricot: A weight-adaptation approach to fixing deep learning models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 376–387.

[53] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. Autotrainer: An automatic dnn training problem detection and repair system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 359–371.

[54] Ziqi Zhang, Yuanchun Li, Yao Guo, Xiangqun Chen, and Yunxin Liu. 2020. Dynamic slicing for deep neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 838–850.

[55] Ziqi Zhang, Yuanchun Li, Jindong Wang, Bingyan Liu, Ding Li, Yao Guo, Xiangqun Chen, and Yunxin Liu. 2022. ReMoS: reducing defect inheritance in transfer learning via relevant model slicing. In *Proceedings of the 44th International Conference on Software Engineering*. 1856–1868.

[56] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009