Element-Based Automated DNN Repair with Fine-Tuned Masked Language Model

XU WANG, Beihang University, China, Zhongguancun Laboratory, China, and Engineering Research Center of Intelligent Computing for Complex Energy Systems, Ministry of Education, China
MINGMING ZHANG, Beihang University, China
XIANGXIN MENG*, Beihang University, China
JIAN ZHANG, Nanyang Technological University, Singapore
YANG LIU, Nanyang Technological University, Singapore
CHUNMING HU, Beihang University, China

Deep Neural Networks (DNNs) are prevalent across a wide range of applications. Despite their success, the complexity and opaque nature of DNNs pose significant challenges in debugging and repairing DNN models, limiting their reliability and broader adoption. In this paper, we propose MLM4DNN, an element-based automated DNN repair method. Unlike previous techniques that focus on post-training adjustments or rely heavily on predefined bug patterns, MLM4DNN repairs DNNs by leveraging a fine-tuned Masked Language Model (MLM) to predict correct fixes for nine predefined key elements in DNNs. We construct a large-scale dataset by masking nine key elements from the correct DNN source code and then force the MLM to restore the correct elements to learn the deep semantics that ensure the normal functionalities of DNNs. Afterwards, a light-weight static analysis tool is designed to filter out low-quality patches to enhance the repair efficiency. We introduce a patch validation method specifically for DNN repair tasks, which consists of three evaluation metrics from different aspects to model the effectiveness of generated patches. We construct a benchmark, $Benchmark_{APR4DNN}$, including 51 buggy DNN models and an evaluation tool that outputs the three metrics. We evaluate MLM4DNN against six baselines on $Benchmark_{APR4DNN}$, and results show that MLM4DNN outperforms all state-of-the-art baselines, including two dynamic-based and four zero-shot learning-based methods. After applying the fine-tuned MLM design to several prevalent Large Language Models (LLMs), we consistently observe improved performance in DNN repair tasks compared to the original LLMs, which demonstrates the effectiveness of the method proposed in this paper.

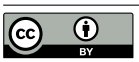
CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Program Repair, Deep Neural Network, Masked Language Model, Fine-Tune

ACM Reference Format:
Xu Wang, Mingming Zhang, Xiangxin Meng, Jian Zhang, Yang Liu, and Chunming Hu. 2025. Element-Based Automated DNN Repair with Fine-Tuned Masked Language Model. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE006 (July 2025), 24 pages. https://doi.org/10.1145/3715716

*Corresponding author: Xiangxin Meng, mengxx@buaa.edu.cn

Authors' Contact Information: Xu Wang, SKLCCSE, Beihang University, Beijing, China and Zhongguancun Laboratory, Beijing, China and Engineering Research Center of Intelligent Computing for Complex Energy Systems, Ministry of Education, Baoding, China, xuwang@buaa.edu.cn; Mingming Zhang, SKLCCSE, Beihang University, Beijing, China, mmzhang@buaa.edu.cn; Xiangxin Meng, SKLCCSE, Beihang University, Beijing, China, mengxx@buaa.edu.cn; Jian Zhang, Nanyang Technological University, Beijing, Singapore, jian_zhang@ntu.edu.sg; Yang Liu, Nanyang Technological University, Beijing, Singapore, yangliu@ntu.edu.sg; Chunming Hu, SKLCCSE, Beihang University, Beijing, China, hucm@buaa.edu.cn.



ACM 2994-970X/2025/7-ARTFSE006
https://doi.org/10.1145/3715716

1 Introduction

Deep Neural Networks (DNNs) have demonstrated outstanding performance in a wide range of fields, such as machine translation (Singh et al., 2017, Stahlberg, 2020), recommendation systems (Da'u and Salim, 2020, Ko et al., 2022), and autonomous driving (Feng et al., 2021, Grigorescu et al., 2020), etc. Particularly, with the popularity of Large Language Models (LLMs) and Artificial Intelligence Generated Content (AIGC) in the most recent years, DNNs have become increasingly crucial and indispensable, serving as the cores of numerous intelligent systems (Wang et al., 2024, Xu et al., 2024). Different from traditional software development, the executions of DNNs are determined by the propagation of weights across complex network structures (Eniser et al., 2019), which makes it challenging to understand and repair DNN bugs (Eniser et al., 2019, Humbatova et al., 2020, Islam et al., 2019, 2020, Ma et al., 2018, Wardat et al., 2021, Zhang et al., 2021). Therefore, automated tools aiding the bug management of DNNs will greatly assist developers in constructing deep learning systems.

In recent years, some methods (Ma et al., 2018, Sohn et al., 2023, Zhang and Chan, 2019) have been proposed to repair buggy neurons in the trained DNN models. For instance, Apricot (Zhang and Chan, 2019) employs a novel weight-adaptation method, improving a DNN model by adjusting the original weights based on the reduced models trained on different subsets of datasets. However, as mentioned in DeepFD (Cao et al., 2022), since these techniques regard DNNs as black boxes, the root causes of undesirable behaviors in DNNs, such as inappropriate activation functions, may be obscured for developers, which cannot facilitate pinpointing the bugs in deep learning programs. Thus, there are several recent studies (Cao et al., 2022, Schoop et al., 2021, Wardat et al., 2021) focusing on detecting and localizing bugs in a more explicit way, such as exposing the inappropriate configurations for DNN training. Despite this, substantial effort is required for developers to comprehend the bug information and to manually create patches for DNN bugs.

To that end, several repair methods are proposed to assist developers in repairing DNN models. AUTOTRAINER (Zhang et al., 2021) is an automated detection and repair system for DNN training problems, which detects bugs by monitoring and analyzing the data collected during model training. Based on the detected symptoms (e.g., vanishing gradient), AUTOTRAINER applies the predefined solutions (e.g., substituting activation functions) to fix them. Similar to AUTOTRAINER, DeepDiagnosis (Wardat et al., 2022) applies the dynamic analysis during model training to detect bugs, which reports actionable fix suggestions by applying the predefined decision trees. Based on the suggestions, such as "Change the activation function at layer: 1", the developer can fix the DNN model. In summary, both of them process the DNN training programs and pinpoint more accurate root causes of the bugs, which are more practical to assist developers in comprehending and repairing bugs.

However, these dynamic approaches exhibit three primary limitations. **First**, the bug repair relies on predefined rules, which are unable to solve the bugs whose types are beyond the capability of these rules. **Second**, the deep semantic features embedded in DNN source code are not leveraged. The typical structure of DNN code includes data preparation, model architecture definition, and model training. The code snippets related to dataset preparation (called *data context*) generally encompass valuable information regarding the types of learning tasks. For example, if a program prepares a dataset by converting output values from strings to zeros/ones (e.g., "positive" $\rightarrow$ 1 and "negative" $\rightarrow$ 0), this suggests that the program is more likely to deal with a binary classification problem. Such insights are crucial for selecting appropriate activation functions and loss functions. Similarly, the code snippets related to model architecture definition and training (called *model context*) encompass more detailed information regarding model design and training configurations. Thus, we believe that the static semantics embedded in the source code have a chance to be utilized

to further enhance the performance of DNN repair tasks. **Third**, these methods generate patches or recommend fixes by analyzing the model runtime information, which cannot be produced before time-consuming model training to achieve rapid iterations in DNN bug management.

On one hand, to address the first two limitations, an effective learning-based method is needed. However, as mentioned in Cao et al. (Cao et al., 2022), there is no existing off-the-shelf dataset available for learning the semantic features of DNN bugs. Additionally, AlphaRepair (Xia and Zhang, 2022) points out that gathering bug-fix pairs via manually crafted rules imposes limitations on the data collection processes, which restricts the repair performance of learning-based methods designed on this basis. Consequently, it introduces *cloze* tasks to the Automated Program Repair (APR) research area. This approach utilizes Masked Language Models (MLMs) and zero-shot paradigms to predict appropriate code elements for repair tasks, achieving promising results (Xia and Zhang, 2022). Nevertheless, most of the existing pre-training LLMs are not directly designed for APR tasks, thus the knowledge learned from the pre-training phase may not be associated with bug-related semantic features, which further limits the repair effectiveness of such methods based on zero-shot paradigms. Recent studies have demonstrated that the fine-tuning paradigm is more effective for achieving better APR performance (Huang et al., 2023, Jiang et al., 2023). These existing methods require bug-fix pairs to learn the underlying repair semantics. Nevertheless, constructing such datasets for DNN repair is challenging (Cao et al., 2022), making it difficult to realize the training process from bug code snippets to the fixed code snippets. To address this limitation, we have gathered adequate DNN code samples with correct functionalities, to fine-tune the model via the format of cloze tasks, enabling it to learn correct code elements. This allows the model to effectively replace erroneous elements when they occur. **On the other hand, to address the third limitation**, we do not utilize the dynamic features extracted from model training phases to generate patches; instead, we follow existing methods focusing on APR tasks in traditional software (Huang et al., 2023, Meng et al., 2023, Xia et al., 2023a, Xia and Zhang, 2022, 2024), which relies solely on the static source code.

Based on the above insights, we propose an element-based DNN repair approach, MLM4DNN, which fine-tunes an MLM to learn deep semantic features of DNNs and generates patches to repair them by predicting correct elements. Specifically, unlike previous learning-based APR methods for traditional software development (Chen et al., 2021, Gupta et al., 2017, Li et al., 2022b, Meng et al., 2022, 2023, White et al., 2019, Ye et al., 2022, Zhu et al., 2021), which train models on collected bug-fix pairs, we collect DNN samples from the latest commits and transform APR tasks to *cloze* tasks (i.e., masking and infilling key elements in DNN source code), thereby promoting the model to learn code elements capable of expressing correct DNN functionalities. Specifically, we define 9 key element types. Then, we construct a dataset tailored for element prediction tasks from the latest commits of real-world repositories, and fine-tune a pre-trained MLM (i.e., UniXcoder (Guo et al., 2022)), to learn to predict correct elements for the masked locations. Based on the prediction results of *cloze* tasks, real patches are generated by replacing the "<mask>" placeholder with the top-ranked predicted elements, which are generally called patch candidates. Furthermore, to improve the efficiency, we design a static checking tool to filter out low-quality patches by analyzing differences between generated patches and original DNN programs. Additionally, inspired by the automated patch validation phase used in APR methods for traditional software development, we design a patch validation method tailored for DNN repair tasks. Furthermore, building on previous work (Cao et al., 2022), we construct a benchmark called $Benchmark_{APR4DNN}$ (51 DNN samples in total), in which an automation tool is designed for systematic evaluations on DNN repair methods.

We conduct extensive experiments on $Benchmark_{APR4DNN}$ to compare MLM4DNN with six baselines, including two dynamic-based DNN repair methods and four zero-shot learning-based methods. Our method significantly outperforms all baselines across three metrics. Specifically,

MLM4DNN (1) significantly improves the performance[1] of 88% DNN samples, outperforming all baselines by more than 36.36%; (2) enables over 70% DNN samples to reach the developer-desired performance, outperforming all baselines by more than 71.43%; (3) successfully generates patches that are semantically matched with the ground truth for 49.02% DNN samples, outperforming all baselines by more than 150%. Additionally, we conduct extensive ablation studies, demonstrating the effectiveness of five main designs of MLM4DNN. We also evaluate the effectiveness of MLM4DNN with different base models (i.e., UniXcoder (Guo et al., 2022), CodeT5 (Wang et al., 2021), InCoder (Fried et al., 2023), Llama 3 (Touvron et al., 2023), DeepSeek-Coder (Guo et al., 2024), GPT-3.5 (OpenAI, 2022), and GPT-4o (OpenAI, 2024)), and study whether these base models could achieve better results after incorporating our approach. Experimental results show that repair results of each base model are significantly better than those where our approach is not leveraged, further proving the effectiveness of our method in DNN repair tasks.

The main contributions of this paper are as follows:

- **Direction.** This paper is the first to propose a DNN repair method that relies solely on DNN source code, which repairs buggy DNNs by revisiting the *cloze-style* APR method within the DNN repair domain. Additionally, we enhance the repair performance by fine-tuning an MLM on DNN samples collected from the latest commits of open-source projects. This work opens a new direction for DNN repair tasks, utilizing cloze-style design in source code and fine-tuning paradigm to boost repair performance.
- **Technique.** We propose MLM4DNN, an element-based DNN repair approach that repairs buggy DNNs by predicting masked elements, enhanced by fine-tuning base models on a large-scale dataset (not bug-fix pairs) constructed from open-source repositories and employing a static checking tool to filter out low-quality patches.
- **Benchmark & Evaluation.** We construct $Benchmark_{APR4DNN}$, which includes 51 buggy DNN samples and an evaluation tool that validates & categorizes patches and outputs three key metrics (i.e., WRC, SRC, and SMC), thereby enabling systematic and quantitative evaluations for DNN repair methods.
- **Extensive Study.** We conduct extensive experiments by comparing MLM4DNN with six methods (both dynamic-based and zero-shot learning-based methods) on $Benchmark_{APR4DNN}$, demonstrating that our approach significantly outperforms all baselines. We also evaluate its effectiveness across various base models and find that MLM4DNN with UniXcoder achieves the best overall performance. Additionally, MLM4DNN surpasses methods using the same LLMs alone, confirming the effectiveness of our design for DNN repair tasks.

2 Related Work

2.1 Debugging for Deep Neural Networks

Several studies focus on the ill-trained DNN models. Apricot (Zhang and Chan, 2019), GenMuNN (Wu et al., 2022) and NNRepair (Usman et al., 2021) adjust weights in DNNs to fix models, while MODE (Ma et al., 2018) and DeepFault (Eniser et al., 2019) fix the suspicious neurons based on model retraining. HybridRepair (Li et al., 2022a) offers a solution using both annotated and unlabeled data for repairs. They focus on fixing model weights after training is complete. Unlike these, MLM4DNN fixes bugs in the DNN training code directly. Additionally, SENSEI (Gao et al., 2020) aims to fix DNNs by augmenting training data, which focuses on image-specific tasks with geometric and color operations, making it inapplicable to other tasks (e.g., NLP). Unlike it, MLM4DNN repairs DNNs by adjusting their code, enabling it to be generalized beyond a specific

[1]In this paper, "performance" is used to refer the accuracy or loss values of a DNN model, as used in the previous study (Zhang et al., 2022).

task type. Recent research, such as DeepLocalize (Wardat et al., 2021), Neuralint (Nikanjam et al., 2022), UMLAUT (Schoop et al., 2021), and DeepFD (Cao et al., 2022), focuses on identifying and localizing bugs in training programs, with methods ranging from static and dynamic analysis to learning-based fault localization. Another method, deepmufl (Ghanbari et al., 2023), revisits mutation-based bug localization in DNN models. In contrast, MLM4DNN aims to automatedly repair bugs to improve the performance of DNN models. AUTOTRAINER (Zhang et al., 2021) particularly focuses on detecting and repairing five common training bugs by applying predefined solutions based on the data recorded from dynamic monitoring. DeepDiagnosis (Wardat et al., 2022) is an automated bug diagnosis method, which provides actionable fix suggestions based on the detected symptoms to assist user in repairing DNNs. The two methods are rule-based methods, which cannot identify and repair the bugs that do not conform to these predefined patterns. Our approach, by learning how to predict the correct elements from a vast amount of static data, can overcome these shortcomings. Schumi et al. (Schumi and Sun, 2023) propose a semantic-based method for fixing bugs related to DNN layer configurations. In contrast, our approach improves DNN performance by correcting training configurations rather than just fixing layer bugs.

2.2 APR for Traditional Software Development

Template-based APR methods synthesize patches by utilizing predefined fix templates (Koyuncu et al., 2020, Liu et al., 2019, Meng et al., 2022, Saha et al., 2019), which generally provide good repair performance for bug types covered by templates, yet may not produce effective patches for those beyond their scope. There are some other learning-based methods (Chen et al., 2021, Li et al., 2022b, Meng et al., 2023, Ye et al., 2022, Zhu et al., 2021) that treat the APR task as a statistical machine translation task. These learning-based APR methods utilize training datasets constructed from bug-fix commits, where each sample consists of a pair of a bug and its corresponding fixed code. In contrast, we construct our training dataset by collecting DNN code from the most recent commit of each repository, ensuring all historical bug-fix commits are applied, where each sample consists of a pair of a DNN containing a *<mask>* placeholder and its corresponding original element. Recently, the advent of LLMs has opened up new opportunities for APR research (Feng et al., 2020, Guo et al., 2022, Huang et al., 2023, Jiang et al., 2023, OpenAI, 2022, Wang et al., 2021). AlphaRepair (Xia and Zhang, 2022) leverages a pre-trained MLM to predict the correct code for bug locations with context based on CodeBERT (Feng et al., 2020) and zero-shot learning. GAMMA (Zhang et al., 2023) revisits template-based APR via mask prediction, which utilizes the UniXcoder base model to predict the correct code elements masked by predefined fix templates. Unlike them that focus on repairing bugs in traditional software programs, MLM4DNN is specifically designed to address issues in DNN models. Furthermore, different from them only leveraging pre-trained MLMs, MLM4DNN employs a fine-tuning paradigm to leverage domain-specific knowledge related to DNNs, thereby enhancing repair performance.

3 Approach

3.1 Overview

In this work, we propose MLM4DNN, an element-based APR method for DNNs powered by a fine-tuned MLM. It generates patches for buggy DNN models by predicting the masked suspicious code elements. As shown in Figure 1, MLM4DNN mainly includes four components. The first component (Section 3.2) constructs a large-scale element prediction dataset from the latest commits of top-rated open-source repositories and fine-tunes a pre-trained MLM to learn for element prediction. The second component (Section 3.3) synthesizes patch candidates based on the element prediction model. The third component (Section 3.4) proposes a post-processing tool designed to filter out

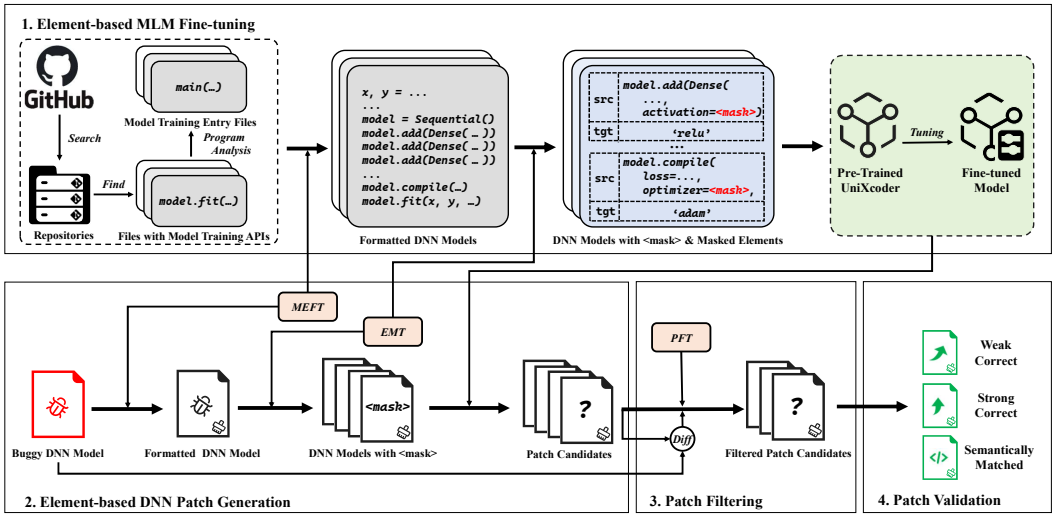


Fig. 1. An Overview of MLM4DNN

Table 1. DNN Elements and the Corresponding Masking Actions

Categories	Name	Usage Forms in Keras	Element Masking Actions
Model Structure	Activation	1. "<activation_name>" 2. keras.activations.<activation_name>	-...keras.layers...(..., activation=<activation>, ...)... +...keras.layers...(..., activation=<mask>, ...)...
	Initializer	1. "<initializer_name>" 2. keras.initializers.<initializer_name> 3. keras.initializers.<initializer_name>(...)	-...keras.layers...(..., kernel_initializer=<initializer>, ...)... +...keras.layers...(..., kernel_initializer=<mask>, ...)...
	Layer	1. keras.layers.<layer_name>(...)	-...keras.layers...(...)... +...<mask>...
Model Compilation	Loss	1. "<loss_name>" 2. keras.losses.<loss_name> 3. keras.losses.<loss_name>(...)	-...compile(..., loss=<loss>, ...)... +...compile(..., loss=<mask>, ...)...
	Optimizer	1. "<optimizer_name>" 2. keras.optimizers.<optimizer_name>(...)	-...compile(..., optimizer=<loss>, ...)... +...compile(..., optimizer=<mask>, ...)...
	Learning Rate	1. a number	-...keras.optimizers...(..., learning_rate=<learning_rate>, ...)... +...keras.optimizers...(..., learning_rate=<mask>, ...)...
	Metric	1. "<metric_name>" 2. keras.losses.<metric_name> 3. keras.losses.<metric_name>(...)	-...compile(..., metrics=<metrics>, ...)... +...compile(..., metrics=<mask>, ...)...
Model Training	Batch Size	1. an integer	-...fit(..., batch_size=<batch_size>, ...)... +...fit(..., batch_size=<mask>, ...)...
	Epochs	1. an integer	-...fit(..., epochs=<epochs>, ...)... +...fit(..., epochs=<mask>, ...)...

low-quality patch candidates by analyzing differences between patches and original DNN models. The final component is utilized to evaluate the effectiveness of filtered patch candidates.

3.2 Element-Based MLM Fine-Tuning

3.2.1 DNN Key Element Definition. DNN models in Keras (Chollet, 2015) generally comprise three main parts: model structure definition, model compilation (*model.compile(...)*), and model training (*model.fit(...)*). We refer to DNN repair patterns proposed in Islam et al. (Islam et al., 2020) and Zhang et al. (Zhang et al., 2021), which primarily focus on three categories of DNN bugs, including model-related, dataset-related, and API-related bugs. Among them, the repair patterns for model-related bugs attract increasing interest (Cao et al., 2022, Ghanbari et al., 2023, Zhang et al., 2021), which

Algorithm 1 Element Masking of Buggy DNN

Input: DNN M, masking actions A;
Output: List of (DNN with <*mask*>, Original element) L;

```
L ← [];
for each a ∈ A do
    target_elements ← get_target_elements(M, a);
    for each e ∈ target_elements do
        M_mask ← replace_element_with_mask(M, e);
        L ← L + [(M_mask, e)];
    end for
end for
return L;
```

can be further divided into two sub-groups. The first involves correcting sub-optimal settings (e.g., activation functions and learning rate), while the second involves revising inappropriate tensor shapes to ensure the correct connections between adjacent layers. For the latter, since the concerned bugs directly lead to the inability to perform compliant forward and backward propagation (Islam et al., 2020, Schumi and Sun, 2023), they are not considered problems in the DNN training phase (Cao et al., 2022, Ghanbari et al., 2023, Schumi and Sun, 2023, Zhang et al., 2021), and therefore, are not within the scope of this paper. Based on the remaining patterns, including correcting the loss(1) and activation functions(2), altering the network layers(3), establishing appropriate metrics(4), modifying the batch size(5) and training epochs(6), adjusting the optimizer(7) and learning rate(8), as well as setting a proper initializer(9), we identify 9 key elements in DNN training phases. From different functional stages of DNNs, three of them focus on the model structure (*Activation*, *Initializer*, *Layer*), four of them concentrate on the configurations for model compilation (*Loss*, *Optimizer*, *Learning Rate*, *Metric*), while the remaining two elements are related to model training (*Batch Size*, *Epochs*).

Afterwards, for each element, we analyze and summarize the usage forms in Keras, as well as operations involving element masking (as called element masking action in this paper). As shown in Table 1, we list the usage forms in Keras and element masking actions for all DNN elements defined in this paper. To generate DNN models containing a <*mask*> (as called the set of DNN_m in this paper), we define element masking actions for all elements. Based on these actions, we develop a syntax-based element masking tool (EMT), as detailed in Algorithm 1. Specifically, given a DNN model M, masking actions A, for each action in A, EMT searches for the target elements required by the action and replaces each target element with a "<mask>" token.

3.2.2 Dataset Construction. Numerous high-quality DNN models implemented in Keras are needed by our approach to a train element prediction model. To obtain sufficient DNN models, we first download 5,178 repositories from GitHub that have more than 10 stars by searching with the keyword "keras". Then, we check whether any of the collected repositories are also present in the benchmark used for experiments in this paper to avoid data leakage, and find that no such cases exist. Subsequently, for each repository, we extract DNN models from the latest commits of the main branches by analyzing python files in them. Since large quantities of projects utilize jupyter notebooks [2] for DNN model construction and training, all notebook files (i.e., files with the ".ipynb" extension) in the collected repositories are converted to Python files via the built-in command "jupyter nbconvert". Afterwards, the process of these Python files can be divided into three steps. First, we search for all Python files containing at least one of the model training APIs,

[2] https://jupyter.org

Table 2. Statistics of $Dataset_{MLM}$ and the Corresponding Standard Usage Forms for Nine Pre-defined Key Elements

Elements	#Samples	Standard Usage Forms
Activation	36,330	"<activation_name>"
Initializer	9,441	"<initializer_name>"
Layer	29,279	keras.layers.<layer_name>
Loss	8,761	"<loss_name>"
Optimizer	9,031	keras.optimizers.<opti_name>()
Learning Rate	4,499	<a number>
Metric	5,684	"<metric_name>"
Batch Size	4,149	<an integer>
Epochs	5,601	<an integer>
Total	**112,775**	-

including *model.fit(...)*, *model.fit_generator(...)*, and *model.train_on_batch(...)*. As a result, 15,047 files are collected. Next, we build the call graph for these projects using pyan [3] , an offline call graph generator for Python. Based on the call graphs, we find the root callers of these training APIs and the corresponding entry files. In this step, we collect 12,169 model training entry files.

As mentioned in GAMMA (Zhang et al., 2023), the precision of element prediction is quite limited when only a single masked buggy line is given without any context. Therefore, we develop a *model extracting and formatting tool (MEFT)* to extract DNN models with more useful context and format the obtained DNN code. Given an entry file, we first inline the module imports and function calls, extracting the code of the module or function into the importing or calling point. Note that MEFT only considers modules/functions defined in the project that correspond with the entry file. Then we discard the statements that are unrelated to model training by applying program slicing. Afterwards, the DNN models with full context are extracted, including the data context and model context. The former contains the statements related to dataset preparation, while the latter comprises the statements about model structure definition, model compilation, and model training. In this paper, both contexts are collaboratively employed for the training and inference phases of the element prediction model. The following Nevertheless, due to the dynamic language feature of Python, the previous steps may fail in certain scenarios, the files in which are discarded. Finally, duplicate samples from the collected models are removed, after which there are 7,376 DNN models extracted in total. All above operations are applied to Abstract Syntax Trees (ASTs) of DNN models. Then, we obtain formatted DNN code by utilizing *ast.unparse(...)*. Finally, EMT is applied to the obtained formatted DNN models, after which the mask-infilling dataset $Dataset_{MLM}$ is constructed, consisting of 112,775 samples (the detailed statistics are shown in Table 2). Each sample is the pair of a DNN_m and the corresponding original element.

3.2.3 Fine-Tuning. We select UniXcoder(Guo et al., 2022), an MLM, as base model, which has been demonstrated to be effective in APR tasks across both zero-shot (Zhang et al., 2023) and fine-tuning paradigms (Huang et al., 2023), and fine-tune it on $Dataset_{MLM}$. UniXcoder is a unified cross-modal pre-trained model for code, which utilizes the mask attention mechanism with prefix adapters to control the model behaviors and leverages cross-modal contents such as ASTs and code comments to further enhance the repair performance.

[3] https://github.com/davidfraser/pyan

```
...
model = keras.models.Sequential()
model.add(Dense(units=8, input_dim=8, init='normal', activation='relu'))
model.add(Dense(units=8, init='normal', activation='relu'))
model.add(Dense(units=8, init='normal', activation='sigmoid'))
sgd = keras.optimizers.SGD(learning_rate=0.01)
model.compile(optimizer=sgd, loss='mean_squared_error', metrics=['acc'])
model.fit(X, y, batch_size=1, epochs=1000, validation_data=(X_test, y_test))
```

(a) Formatted DNN Buggy Model

```
#1
model.compile(optimizer=sgd, loss='mean_squared_error', metrics=<mask>)
#4 ...
model.compile(optimizer=sgd, loss=<mask>, metrics=['accuracy'])
#15 ...
model.add(Dense(units=8, init=<mask>, activation='sigmoid'))
```

(b) DNN Models with <*mask*>

```
#1
model.compile(optimizer=sgd, loss='mean_squared_error', metrics=['acc'])
#4 ...
model.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['acc'])
#15 ...
model.add(Dense(units=8, init='normal', activation='sigmoid'))
```

(c) Patch Candidates

```
#1 (#2 of patch candidates)
sgd = keras.optimizers.SGD(learning_rate=0.1)
#2 (#4 of patch candidates)
model.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['acc'])
#3 (#12 of patch candidates)
model.add(keras.layers.Dense(units=8, init='normal', activation='relu'))
```

(d) Filtered Patch Candidates

Fig. 2. An Example of DNN Model Repair

As mentioned in Section 3.2.2, the original DNN code snippets are transformed to the formatted representations containing the placeholder <*mask*> and the original elements. For our training task, the input is DNN_m, and the expected output is the corresponding original element. Since the length of model input may exceed the max input length of UniXcoder, we split the full input into two segments (i.e., *pre-context* and *post-context*) based on the symbol <*mask*>, and save the most nearest code text in a length constraint $\lfloor max\_input\_length/2 \rfloor$ for each segment. After fine-tuning, we obtain a model capable of predicting the masked element in a DNN_m.

3.3 Element-Based DNN Patch Generation

Given a buggy DNN model, we use our element prediction model to generate patch candidates for it. Compared with existing studies (Cao et al., 2022, Ghanbari et al., 2023, Wardat et al., 2022, 2021, Zhang et al., 2021) that collect the dynamic features during model training, we only employ the static code features for patch synthesis. Unlike the prior automated DNN repair method (Zhang et al., 2021), we are the first to propose a way to repair bugs at the source code level. Following

prior APR methods for traditional programs by considering bug-related features at the source code level (Li et al., 2024, Lutellier et al., 2020, Wei et al., 2023, Ye et al., 2022, Zhu et al., 2021), MLM4DNN focuses on fixing single-hunk bugs, where the patch is obtained by changing a DNN element. Additionally, we analyzed DNN bugs collected by prior work (Cao et al., 2022) from GitHub and Stack Overflow, of which 43.75% are single-hunk bugs. In comparison, as mentioned in the existing study (Xia et al., 2023b), 39.39% of the 391 bugs in Defects4J v1.2.0 and 36.30% of the 438 bugs in Defects4J v2.0.0 are single-hunk bugs, both lower than that of the DNN repair domain. Despite this, many previous studies mentioned above have concentrated on single-hunk bugs, which also provide valuable experience for future research. We divide the patch generation process into three steps, as shown in Figure 1.

(1) Format DNN models. As mentioned in Section 3.2.2, the given buggy model needs to be formatted using MEFT.

(2) Generate DNN_m set. Based on the formatted models, EMT (mentioned in Section 3.2.1) is applied to generate the DNN_m set.

(3) Generate patch candidates. In previous steps, the APR tasks for DNN models are transformed to cloze tasks. For each DNN_m generated by EMT, the top-1 prediction result from our element prediction model is retained.

We provide an example in Figure 2, where only the key contexts of the model code are provided due to space constraints. The formatted buggy model M is shown in Figure 2a. There are totally 15 DNN_m generated by EMT from M. Next, 15 patch candidates are generated for each DNN_m by querying the element prediction model. As shown in Figure 2c, the 4^{th} patch updates the loss function from 'mean_squared_error' to 'binary_crossentropy', while the other two displayed patches are the same as the original DNN model.

3.4 Patch Filtering Based on Patch Differential Analysis

To further improve the repair performance, a patch filtering tool (PFT) is developed to filter out the low-quality patches by analyzing the differences between patch candidates and the corresponding original buggy DNN models. This tool is capable of filtering out three types of patches:

- The patches with syntax errors.
- The patches that are semantically equivalent to original DNN models.
- The patches without appropriate input and output shapes.

Since code changes only occur between the original masked element E_o and the predicted element E_p, PFT exclusively analyzes these two elements. **First**, if E_p has any syntax error, PFT will discard the patch. PFT checks whether E_p is syntactically correct by utilizing the Python built-in library *ast*. If E_p can be successfully parsed into an Abstract Syntax Tree (AST) by *ast.parse*, it is considered to be free of syntax errors. Note that a patch without syntax error may not be executable due to the dynamic language feature of Python (e.g., a non-existing variable is referenced). **Second**, if E_p is semantically equivalent to E_o, the patch is also discarded. As shown in Table 1, for each element type, there may be several different usage forms. For instance, *'mse'*, *keras.losses.MeanSquaredError(...)*, *keras.losses.mean_squared_error* and *'mean_squared_error'* all denote the mean squared error loss function. Thus, according to the Keras document (Chollet, 2015), we develop a component for PFT, which is used to transform the given DNN element to the corresponding standard usage form. We list the standard forms for all elements in Table 2. We take the algorithm for the standardization process of *loss* element as an example, which is shown in Algorithm 2. The *loss* elements have three usage forms as shown in Table 1, which are unified into the form of strings in MLM4DNN. In the algorithm, given a non-standard *loss* element, if it is used in form of class conduction (e.g., *keras.losses.MeanSquaredError(...)*), loss name is obtained from

Algorithm 2 *loss* Element Standardization

Input: Non-standard *loss* element e;
Output: Standardized *loss* element e_{std};

```
loss_name ← None;
if is_function_call(e) then
    // example 1: keras.losses.MeanSquaredError(...)
    loss_name ← get_function_name(e)
else if is_attribute(e) then
    // example 2: keras.losses.mean_squared_error
    loss_name ← get_attribute_name(e)
else if is_string_constant(e) then
    // example 3: 'mse' / 'MSE' / 'mean_squared_error'
    loss_name ← get_string_literal(e)
end if
if loss_name and is_builtin_loss_name(loss_name) then
    loss_name ← short_name_to_full_name(loss_name);
    loss_name ← full_name_to_snake_name(loss_name);
    e_std ← make_string_constant(loss_name);
else
    e_std ← e;
end if
return e_std;
```

class name. And if is used in form of attribute (e.g., *keras.losses.mean_squared_error*), loss name is obtained from attribute name. Then, loss name is converted from short name (e.g., *'mse'*) to full name (e.g., *'mean_squared_error'*), and then is converted from full name (e.g., *'MeanSquaredError'*) to snake name (e.g., *'mean_squared_error'*). PFT transforms E_p and E_o to their standard forms $E_p^{\#}$ and $E_o^{\#}$. If $E_p^{\#}$ equals $E_o^{\#}$, the patch is filtered out. **Third**, if the input shape or output shape of the generated patch is inappropriate, it will be discarded. PFT first extracts the input and output shapes of the layer L_{mask} corresponding to E_p and E_o. If the shape of L_{mask} is not changed, the patch will be retained directly. Next, PFT determines whether the shape modification of this layer affects the shapes of the overall network, based on the contextual information of L_{mask} (e.g., the index of L_{mask} and the type of previous/next layer of L_{mask}). For instance, if the L_{mask} is the penultimate layer and its output shape is changed, and if the final layer employs a ReLU activation function (which maintains the same output shape as its input), then the overall output shape of the network will also change. For cases like these, as mentioned in Section 3.2.1, we do not care the bugs related to shapes, hence if the input or output shape of the DNN changes, the patch will be filtered out.

Overall, this tool is used to improve validation efficiency by simply eliminating redundancy (e.g., generating the same element) and obvious issues (e.g., generating the pacthed DNN with error input/output shape) in the final patch candidates. Also taking Figure 2d as an example, it is shown that 12 patches are discarded in the patch filtering process in total. Consequently, only 3 patches remain, which will subsequently proceed to the patch validation stage.

3.5 Patch Validation

Inspired by studies related to APR for traditional software development (Ghanbari, 2020, Ghanbari and Marcus, 2022, Liu et al., 2019, Meng et al., 2022, Xia et al., 2023b, Xia and Zhang, 2022, Zhang et al., 2023), a patch validation process is needed to evaluate the DNN repair performance. However, due to the stochastic nature of DNN model training and inference, traditional test-driven patch validation

methods fall short, as they are designed for deterministic scenarios and cannot accommodate the inherent output variability. Therefore, we design a patch validation method specifically for DNN repair tasks, which cares about three distinct types of patches:

- **Weak Correct Patch**: Patches that demonstrate a statistically significant improvement in performance compared to the original buggy model.
- **Strong Correct Patch**: Patches whose performance is greater than or equal to that of the corresponding ground truth model.
- **Semantically Matched Patch**: Patches whose source code is semantically equivalent to the corresponding ground truth model.

Specifically, we perform multiple rounds of training for each buggy model, ground truth model, and generated patch candidate first. This repetition helps to minimize the impact of randomness on the evaluation results.

To determine if a patch qualifies as a weak correct patch, we first check whether there is a significant statistical difference between the distribution of the performance of the patched DNN (i.e., $PD_{\mathcal{P}}$, where PD indicates the performance of DNN) and that of the original buggy DNN PD_O using the following equation (Humbatova et al., 2021, Jahangirova and Tonella, 2020):

$$hasSigDiff(O, \mathcal{P}, X) = \left\{ \begin{array}{ll} true & \text{if effectSize}(PD_O(X), PD_{\mathcal{P}}(X)) \geq \beta \\ & \text{\& pValue}(PD_O(X), PD_{\mathcal{P}}(X)) < \alpha \\ false & \end{array} \right\} \quad (1)$$

where α and β are thresholds that control the statistical significance and effect size, while X represents the testing set. We employ the generalized linear model (GLM) (Nelder and Wedderburn, 1972) for the calculation of statistical significance and Cohen's d (Cohen, 1992) for the effect size. If the *accuracy* distribution of the patched DNN significantly differs from that of the original model, we further examine whether the average accuracy of the patched DNN surpasses that of the original one. A patch is classified as weak correct if both conditions are satisfied.

To determine if a patch qualifies as a strong correct patch, we first calculate the average performance. If the patch performance is equal to or exceeds that of the ground truth, it is considered strong correct. Additionally, if a patch's performance is marginally lower than the ground truth but does not exhibit a significant difference (as determined by Equation 1), we still consider the patch to be on par with the ground truth model.

To determine if a patch qualifies as a semantically matched patch, we standardize all elements in both the patch and the ground truth model as outlined in Section 3.4. If the transformed source code is equivalent to that of the ground truth model, the patch is deemed a semantically matched patch.

By analyzing the presence of these three patch types within the generated patch candidates for each buggy model, we can comprehensively evaluate the performance of the DNN repair methods. Specifically, we use the following three metrics, corresponding to the three types of patches.

(1) Weak Repair Count (WRC): This metric represents the number of DNNs whose corresponding patch candidates contain at least one weak correct patch. This metric reflects the method's ability to generate patches that significantly improve model performance, with each DNN considered weakly repaired if it contains at least one weak correct patch.

(2) Strong Repair Count (SRC): This metric quantifies the number of DNNs whose corresponding patch candidates contain at least one strong correct patch. This metric evaluates whether the generated patches can achieve performance levels comparable to the ground truth, indicating that a DNN is strongly repaired if it includes at least one strong correct patch.

(3) Semantically Matched Count (SMC): This metric indicates the number of DNNs whose patch candidates contain at least one semantically matched patch. This metric assesses the acceptability of the generated patches from the developers' perspective, reflecting whether the repair method can produce solutions that are fully aligned with the developer's intent.

4 Evaluation

4.1 Research Questions

In this paper, we study the following research questions:

- **RQ1:** How does MLM4DNN perform in DNN repair compared to existing techniques?
- **RQ2:** How do the main designs of proposed MLM4DNN impact its overall effectiveness?
- **RQ3:** How does MLM4DNN perform across different base models, and how does it improve performance compared to using models alone?

4.2 Evaluation Metrics

We validate the generated patches according to the approach described in Section 3.5 and evaluate the repair performance of the DNN repair methods using three metrics WRC, SRC, and SMC. Among these metrics, higher values are preferable, indicating improved performance in methods.

4.3 Benchmark Construction

We collect 48 buggy DNN models from the publicly accessible repository of the prior research DeepFD (Cao et al., 2022)[4], which are real-world DNNs implemented in Keras from *Stack Overflow* and *GitHub* and are used by previous studies (Cao et al., 2022, Wardat et al., 2022, 2021). Due to the existence of multiple fixes within a single model, we decompose each bug location and the corresponding repair action into an independent repair task. Through this approach, we obtain a total of 89 test samples containing individual repair behaviors. However, not all samples represent true bugs, as previous studies lack rigorous statistical analyses of their performance. Therefore, we conduct further filtering of these samples. Specifically, for each buggy sample, we train it and its corresponding ground truth model N times and then apply Equation 1 to determine if there is a significant performance difference between the two. If a significant difference is found and the performance of the ground truth model is better than that of the buggy sample, the sample is retained; otherwise, it is discarded. In our experiments, following the previous studies (Cao et al., 2022, Humbatova et al., 2021), which also employ this formula for further checking of possible buggy DNNs, we set N to 20, α to 0.05, and β to 0.2. After filtering, a total of 51 buggy samples are retained. Then, we ensure that these DNNs are not included in our training dataset by querying them in models of the training dataset, in order to avoid data leakage. Subsequently, we develop an automated evaluation tool for this benchmark, based on the patch validation process described in Section 3.5, using the same values for N, α, and β as those employed in the filtering of buggy samples. This tool automatically identifies three types of patches (i.e., weak correct patch, strong correct patch, and semantically matched patch) from the patch candidates for each buggy DNN and outputs the three metrics outlined in Section 4.2.

Finally, we construct the benchmark, denoted as $Benchmark_{APR4DNN}$, including 51 buggy DNN samples and a tool designed to evaluate the effectiveness of DNN repair methods on this benchmark. Through manual analysis, we conclude that the benchmark includes DNNs with two to more than ten layers, incorporating various modules such as dense, convolutional, recurrent, pooling, and

[4]The paper of DeepFD mentions 58 buggy models, but only 48 are available in its public repository: https://github.com/ArabelaTso/DeepFD

dropout layers. And the application domains covered span natural language processing (NLP), computer vision (CV), cybersecurity, physics, botany, and other fields.

4.4 Experimental Settings

First, we split $Dataset_{MLM}$ into training, validation, and test sets by allocating the samples according to the DNNs they originate from via a ratio of 8:1:1. As a result, we obtain the training set with 89728 samples, the validation set with 11620 samples, and the test set with 11427 samples. We set the max_source_length of the UniXcoder to 1000 and the max_target_length to 24, as 90% of the training samples have output lengths of fewer than 24 tokens. The AdamW optimizer (Loshchilov and Hutter, 2019) is adopted with the following detailed setups: $lr = 5e-5$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$. We set random seed as 1234 to enhance the reproducibility, and fine-tune 10 epochs on $Dataset_{MLM}$ to enable the pre-trained UniXcoder model to sufficiently learn deep semantic knowledge for the DNN element prediction task.

All the experiments are conducted on Ubuntu 22.04.4 LTS server with 48 cores of 3.2GHz CPU, 264GB RAM, and NVIDIA Tesla A100 GPUs with 40GB memory.

4.5 Baselines

We select the following DNN repair methods as baselines, including two dynamic-based methods and four zero-shot learning-based methods.

(1) AUTOTRAINER (Zhang et al., 2021). AUTOTRAINER is the state-of-the-art automated DNN bug repair system based on dynamic analysis, supporting five predefined bugs, including vanishing gradient, exploding gradient, dying ReLU, oscillating loss, and slow convergence. For each buggy DNN model, when a problem is detected, the predefined solution for the problem will be applied, and the new model will be retrained until it passes the validation process. We use the default settings provided in its open source repository [5].

(2) DeepDiagnosis (Wardat et al., 2022). DeepDiagnosis is an automated method for diagnosing bugs and recommending actionable fix suggestions for the given DNN programs, which intercepts the training process once a predefined bug pattern is detected. However, it is not designed to directly repair bugs in DNN models. To solve this issue, we manually analyze the actionable bug fix suggestions generated by DeepDiagnosis, which are occasionally coarse-grained. For example, it might recommend changing a certain activation function to a more suitable one without specifying the exact replacement. In our evaluation, we apply a comparatively lenient standard, considering such cases as correct.

Additionally, we design and implement a DNN repair method leveraging the zero-shot learning capabilities of Large Language Models (LLMs), including it as part of our baselines. This broadens the scope of our comparative experiments, and the evaluation results of this zero-shot learning-based method can provide valuable references for future research in the field.

First, following the existing approach (Xia and Zhang, 2024), we set the system prompt for the used LLM to "You are an Automated Program Repair tool" to inform it of its role as an APR tool. Next, we specifically design a prompt consisting of the following three parts: (1) to inform the LLM that we will provide a Keras (Chollet, 2015) model training code snippet containing a bug; (2) to provide the buggy DNN code mentioned above; and (3) to specify requirements aimed at preventing unnecessary output. Before patch filtering, MLM4DNN generates the same number of patches as the detected key elements in the given buggy DNN. To ensure fairness, we follow existing studies (Kolak et al., 2022, Prenner et al., 2022, Xia et al., 2023a,b, Xia and Zhang, 2024) by using sampling method to generate an equivalent number of patches. Following the previous

[5] https://github.com/shiningrain/AUTOTRAINER

Table 3. Comparative Analysis of Repair Results by Different Techniques. Here n.a. means Not Available, and the notation X (Y%) indicates the count X and its corresponding percentage Y

Technique	WRC	SRC	SMC
AUTOTRAINER	6 (11.76%)	2 (3.92%)	0 (0.00%)
DeepDiagnosis[7]	n.a.	n.a.	10 (19.61%)
DeepSeek-Coder	27 (52.94%)	15 (29.41%)	1 (1.96%)
Llama 3	30 (58.82%)	21 (41.18%)	0 (0.00%)
GPT-3.5	33 (64.71%)	20 (39.22%)	2 (3.92%)
GPT-4o	33 (64.71%)	21 (41.18%)	3 (5.88%)
MLM4DNN	**45 (88.24%)**	**36 (70.59%)**	**25 (49.02%)**

studies (Xia et al., 2023a, Xia and Zhang, 2024), we use nucleus sampling with a top-p value of 1 and a temperature of 1 in order to get a diverse set of patches. All patches generated for a buggy DNN are saved as a candidate patch set for subsequent patch validation. It is important to note that the bug locations are not provided, ensuring that the same configuration is used for MLM4DNN as well as for other baselines.

We implement this method with a series of instruction-tuned LLMs that achieve good performance on code-related tasks. (1) **Llama 3 (Touvron et al., 2023)**, which is designed for a wide range of NLP tasks, including coding-related activities. In our experiment, we use Llama-3-8B-Instruct. (2) **DeepSeek-Coder (Guo et al., 2024)**, an LLM trained on a real-world code corpus, which enhances its proficiency in coding-related tasks, including debugging. We employ DeepSeek-Coder-6.7B-Instruct. (3) **GPT-3.5 (OpenAI, 2022)** and **GPT-4o (OpenAI, 2024)**, which are pre-trained LLMs developed by OpenAI [6] and have demonstrated state-of-the-art performance in the APR task (Xia and Zhang, 2024). We access them through the APIs of gpt-3.5-turbo-0125 and gpt-4o-mini-2024-07-18, respectively.

In summary, we compare a total of six baselines, including two dynamic methods (i.e., AUTOTRAINER, DeepDiagnosis) and four zero-shot learning-based methods (i.e., Llama 3, DeepSeek-Coder, GPT-3.5, and GPT-4o).

4.6 Results and Discussion

We investigate the following research questions to provide an analysis of the experimental results.

RQ1: How does MLM4DNN perform in DNN repair compared to existing techniques?

To answer this RQ, we evaluate the repair effectiveness of MLM4DNN and six baseline methods (as detailed in Section 4.5) on $Benchmark_{APR4DNN}$. Table 3 shows the experimental results of these seven methods, including two existing methods specifically designed for DNN debugging (top, i.e., AUTOTRAINER (Zhang et al., 2021) and DeepDiagnosis (Wardat et al., 2022)), four zero-shot learning-based methods (middle, i.e., DeepSeek-Coder (Guo et al., 2024), Llama 3 (Touvron et al., 2023), GPT-3.5 (OpenAI, 2022), and GPT-4o (OpenAI, 2024)), and our method (bottom).

The repair results show that our method outperforms all baseline approaches across three metrics (i.e., WRC, SRC, and SMC). In terms of improving model performance (i.e., WRC), MLM4DNN significantly enhances the performance of 45 out of 51 buggy models, covering over 88% of the

[6] https://openai.com

[7] For DeepDiagnosis, only the SMC is calculated by manually checking if the suggestions match the developers' fixes, as the other two metrics are not applicable to this method.

Table 4. Comparative Analysis with Different Designs for MLM4DNN. Here the notation X (Y%) indicates the count X and its corresponding percentage Y.

Model Variants	WRC	SRC	SMC
MLM4DNN	**45 (88.24%)**	**36 (70.59%)**	**25 (49.02%)**
$Model_{no-pretraining}$	9 (17.65%)	6 (11.76%)	4 (7.84%)
$Model_{no-fine-tuning}$	22 (43.14%)	11 (21.57%)	0 (0.00%)
$Model_{no-data-context}$	45 (88.24%)	33 (64.71%)	16 (31.37%)
$Model_{no-element-masking}$	43 (84.31%)	29 (56.86%)	7 (13.73%)
$Model_{no-patch-filtering}$	45 (88.24%)	36 (70.59%)	25 (49.02%)

samples in $Benchmark_{APR4DNN}$. In contrast, AUTOTRAINER only achieves significant performance improvements in 6 models. Additionally, compared to zero-shot learning-based methods, MLM4DNN outperforms its best competitor, GPT-4o, by 36.36%. In terms of meeting developers' expected performance (i.e., SRC), MLM4DNN enables over 70% of models to reach the desired performance, significantly outperforming AUTOTRAINER and surpassing GPT-4o's success rate of 41.18% by 71.43%. In terms of meeting developers' expectations for source code matching (i.e., SMC), MLM4DNN successfully generates patches that align with the ground truth for nearly half of the buggy models, while all other methods perform poorly on this metric. Among these, DeepDiagnosis is the best-performing method aside from MLM4DNN, successfully providing correct repair suggestions for 10 buggy models (as mentioned in Section 4.5, note that this method does not generate patches directly, but rather offers suggestions).

In summary, our method outperforms the other six methods across all three metrics, particularly demonstrating a significant advantage in the SMC metric. **On one hand**, MLM4DNN outperforms four zero-shot learning-based methods. The results demonstrate that the idea of fine-tuning MLMs on $Dataset_{MLM}$ is able to improve the DNN model repair performance. By systematically masking and reconstructing key elements from correct DNN code, it is possible to make LLMs learn deep semantic features necessary for ensuring the normal functionalities of DNN models. Subsequently, when bugs occur in certain critical components, the well-trained LLMs are capable of predicting the correct elements to repair them, thereby achieving APR implementations for DNN code. **On the other hand**, MLM4DNN also surpasses two existing dynamic-based methods. In contrast to these two methods, MLM4DNN generates patches solely based on the static DNN code, without relying on any dynamic features. The experimental results demonstrate that repairing DNN models using only static information is feasible and can yield outstanding performance by employing a learning-based approach that comprehends deep semantics and generates high-quality patches.

> **RQ1 Summary:** MLM4DNN significantly outperforms all six baseline methods across three key metrics (i.e., WRC, SRC, and SMC) with the element-based fine-tuning and patch generation method, without relying on dynamic features from model training.

RQ2: How do the main designs of proposed MLM4DNN impact its overall effectiveness?

In this RQ, we explore the effectiveness of the five main designs of MLM4DNN. The experimental results are shown in Table 4. The following section provides a detailed discussion of the experimental setup and the analysis of the results.

(1) Without Pretraining. We train the element prediction model for MLM4DNN from scratch on our $Dataset_{MLM}$, using the same architecture and configurations as UniXcoder fine-tuning. The

repair results are presented in the second row of Table 4. This variant can achieve only a WRC of 9 (17.65%), SRC of 6 (11.76%), and SMC of 4 (7.84%). Compared to other variants, MLM4DNN without pretraining phase performs almost the worst across three metrics, except for SMC, where it outperforms only $Model_{no-fine-tuning}$. The experimental results indicate that the pre-trained knowledge in UniXcoder is crucial for MLM4DNN. Training solely on $Dataset_{MLM}$ does not lead to satisfactory performanc e for MLM4DNN. On the other hand, by comparing with $Model_{no-fine-tuning}$, we observe that the knowledge acquired from $Dataset_{MLM}$ alone enables a breakthrough in the SMC metric, highlighting the importance of domain-specific datasets from an indirect perspective.

(2) Without Fine-tuning. We directly utilize the original UniXcoder without fine-tuning to predict the masked elements in this variant. As shown in the third row of Table 4, the $Model_{no-fine-tuning}$ variant outperforms $Model_{no-pretraining}$, achieving a WRC of 22 (43.14%) and SRC of 11 (21.57%), but it performs the worst on SMC, with a value of 0. The results indicate that MLM with substantial pre-trained knowledge can achieve certain effectiveness in DNN repair tasks. However, there remains a significant gap when compared to the complete MLM4DNN. On the other hand, through a comparative analysis of the experimental results from this variant and the $Model_{no-pretraining}$ variant, we can find that the two variants each have their strengths and weaknesses across the three metrics. By combining both approaches, MLM4DNN achieves optimal performance, demonstrating a significant improvement over these two variants, exhibiting a multiplicative enhancement.

(3) Without Data Context. As mentioned in Section 3.2.2, we extract the full context of each DNN model (i.e., data context and model context). To evaluate the effectiveness of this design, we construct this variant. We remove the data contexts from all DNNs in $Dataset_{MLM}$ and then fine-tune a new model over the processed dataset. During the repair phase, the input buggy models from $Benchmark_{APR4DNN}$ undergo the same operation. This variant achieves a WRC of 45 (88.24%), SRC of 33 (64.71%), and SMC of 16 (31.37%). By incorporating the data context, the complete MLM4DNN outperforms this variant by 9.09% on SRC and 56.25% on SMC. The experimental results underscore the importance of the data context (i.e., code snippets related to data loading and preprocessing), which can guide the selection of appropriate model structure, loss functions, and more, thereby enhancing the repair performance of MLM4DNN. In contrast, previous methods (Wardat et al., 2022, Zhang et al., 2021) have primarily focused on either static or dynamic semantic features of model context, ignoring the significance of data context. Our element-based MLM fine-tuning method addresses this gap.

(4) Without Element Granularity Masking. In the implementation of MLM4DNN, we employ a finer-grained masking strategy by masking element-level tokens in the DNN code, as discussed in Section 3.2.1. To evaluate the effectiveness of this design, we implement this variant by modifying the masking granularity to replace the entire buggy code line with the symbol <*mask*>. This coarser-grained masking strategy is applied in both the element-based MLM fine-tuning and the element-based DNN patch generation phases. The variant achieves a WRC of 43 (84.31%), SRC of 29 (56.86%), and SMC of 7 (13.73%). In comparison, the element-based MLM4DNN outperforms this variant by 4.65% on WRC and 24.14% on SRC. More importantly, this design significantly enhances the repair performance, improving SMC by 257%. The results demonstrate that our element-based fine-tuning and repair method can enhance the overall performance of our method.

(5) Without Patch Filtering. As described in Section 3.4, low-quality patches generated during the repair process will be filtered out, allowing only the filtered patch candidates to proceed to the patch validation phase. To evaluate the effectiveness of PFT, we construct this variant by disabling patch filtering, resulting in all generated patches being directly input into the validation phase. The results are listed in the sixth row of Table 4. Notably, this variant exhibits the same values across the three metrics as the complete MLM4DNN, as the discarded patches are not valid patches. In other words, this design only alters the ranks of the patches we care about (i.e., weakly correct, strongly

Table 5. Performance Comparison of MLM4DNN and Zero-Shot-based Method Across Different LLMs. Here the notation X (Y%) indicates the count X and its corresponding percentage Y.

Technique	Model	WRC	SRC	SMC	Tokens
MLM4DNN (with fine-tuning)	UniXcoder (127M)	45 (88.24%)	**36 (70.59%)**	**25 (49.02%)**	-
	CodeT5-Small (60M)	42 (80.39%)	27 (52.94%)	23 (45.10%)	-
	CodeT5-Base (220M)	42 (80.39%)	29 (56.86%)	21 (41.18%)	-
	CodeT5-Large (770M)	45 (88.24%)	33 (64.71%)	21 (41.18%)	-
	InCoder (1B)	45 (88.24%)	**36 (70.59%)**	23 (45.10%)	-
MLM4DNN (with prompting)	Llama 3 (8B)	34 (66.67%)	19 (37.25%)	6 (11.76%)	231,155
	DeepSeek-Coder (6.7B)	37 (72.55%)	23 (45.10%)	4 (7.84%)	373,422
	GPT-3.5 (Closed Source)	37 (72.55%)	22 (43.14%)	11 (21.57%)	236,305
	GPT-4o (Closed Source)	**47 (92.16%)**	33 (64.71%)	14 (27.45%)	235,826
Zero-Shot	Llama 3 (8B)	30 (58.82%)	21 (41.18%)	0 (0.00%)	462,543
	DeepSeek-Coder (6.7B)	27 (52.94%)	15 (29.41%)	1 (1.96%)	721,722
	GPT-3.5 (Closed Source)	33 (64.71%)	20 (39.22%)	2 (3.92%)	454,329
	GPT-4o (Closed Source)	33 (64.71%)	21 (41.18%)	3 (5.88%)	470,148

correct, and semantically matched patches) among the patch candidates for each buggy DNN. We analyze the ranks of the first weak correct patch, strong correct patch, and semantically matched patch among the generated patch candidates for each buggy DNN. For the complete MLM4DNN, the average ranks of three types of patches are 1.73, 2.75, and 2.96, respectively, whereas for this variant, the averages are 2.87, 4.19, and 5.68. The results demonstrate that our PFT can optimize the ranks of these three types of patches, thereby saving time and computational resources during patch validation, which highlights the usability of our patch filtering design.

> **RQ2 Summary:** Pretraining, fine-tuning, data context, and element granularity masking significantly enhance the effectiveness of MLM4DNN, while patch filtering improves validation efficiency without affecting the overall repair effectiveness.

RQ3: How does MLM4DNN perform across different base models, and how does it improve performance compared to using models alone?

To evaluate the effectiveness of MLM4DNN using different base models and investigate whether the performance of these models can be improved by applying our element-based patch generation method, we further conduct more extensive experiments. As shown in Table 5, we replace our base model (i.e., UniXcoder) with four other MLMs, fine-tune new element prediction models, and implement a series of MLM4DNN variants:

(1) **CodeT5 (Wang et al., 2021)**: A model family that follows T5's (Raffel et al., 2020) encoder-decoder architectures, and is pre-trained on several code-specific tasks, including masked tokens prediction. We train three new models based on CodeT5-Small, CodeT5-Base, and CodeT5-Large.
(2) **InCoder (Fried et al., 2023)**: A model family that follows XGLM (Lin et al., 2021)'s decoder-only architecture and is also pre-trained on masked tokens prediction task. We train a new model based on InCoder-1B.

Additionally, we utilize four other instruction-tuned LLMs (i.e., Llama 3, DeepSeek-Coder, GPT-3.5, and GPT-4o), as mentioned in Section 4.5, to perform the element prediction task by directly querying them with prompts adapted from existing work (Zhang et al., 2023).

The experimental results for MLM4DNN and its eight variants are presented in the top and middle sections of Table 5. To further explore the performance improvements that our element-based method offers over using the same LLMs alone, we also include the results of four zero-shot learning-based approaches at the bottom of the table.

(1) Using different models with MLM4DNN. Firstly, based on the results presented in Table 5, MLM4DNN with UniXcoder outperforms other models in terms of SRC and SMC, despite having only 127M parameters. While GPT-4o surpasses UniXcoder in the WRC metric, it falls short on the other two metrics, especially SMC. Among the open-source models, InCoder most closely approaches UniXcoder's overall performance, but its larger parameter size requires more hardware resources during training and inference. Overall, UniXcoder is the most suitable base model for MLM4DNN. Secondly, comparing the three variants based on the CodeT5 models, we observe that performance on the WRC and SMC metrics improves with an increase in parameter size. However, when comprehensively comparing the performance of the five fine-tuned models—UniXcoder, CodeT5 (small, base, and large), and InCoder, with parameter sizes ranging from 60M to 770M—we find that larger parameter models do not necessarily hold a definitive advantage for DNN repair tasks. Thirdly, the experimental results presented in the middle section of Table 5 indicate that the application of prompt-based techniques for DNN repair element prediction within MLM4DNN is feasible. Overall, GPT-4o performs closest to the fine-tuned UniXcoder, followed by GPT-3.5. However, the repair performance of these models is generally inferior to that of the fine-tuned models. Additionally, they either have a larger number of parameters or are closed-source models, which, when considering resource requirements, inference time costs, and security, do not demonstrate a clear advantage.

(2) Using different methods with LLMs. On one hand, the performance of MLM4DNN significantly outperforms that of methods using the same LLM alone, indicating that our element-based patch generation method is effective and can substantially enhance DNN repair performance. On the other hand, we record the total number of input and output tokens for two groups of methods (i.e., the middle and bottom sections of Table 5), as shown in the Tokens column of Table 5. The results demonstrate that the number of tokens consumed by MLM4DNN is significantly lower than that consumed by zero-shot learning-based approaches. This is because our method only requires the prediction of the tokens needed for an element at a time. In local model inference, the number of tokens correlates positively with the consumption of hardware resources. In the case of using closed-source models through remote API calls, the number of tokens consumed directly determines the cost (e.g., OpenAI API charges based on token count [8]). Therefore, our element-based DNN repair method can improve performance while simultaneously reducing hardware and economic costs.

> **RQ3 Summary:** (1) MLM4DNN with UniXcoder achieves the best overall performance compared to variants based on other base models. (2) Larger parameter models do not necessarily offer a definitive advantage for MLM4DNN. (3) While prompt-based fill-in models are usable for MLM4DNN, they do not provide benefits in hardware resource or time efficiency. (4) Our element-based method significantly enhances performance compared to using LLMs alone.

[8] https://openai.com/api/pricing

5 Threats to Validity

External Threat. Due to the lack of the DNN repair dataset, we only conduct experiments on one benchmark, which may not include all DNN architectures and bug types. In the future, we will evaluate the generalizability performance of MLM4DNN on more datasets. Another threat is that we only focus on Keras (Chollet, 2015) in MLM4DNN, despite its alignment with previous studies (Cao et al., 2022, Ghanbari et al., 2023, Wardat et al., 2022, 2021, Zhang et al., 2021). However, since DNNs implemented across different libraries often share similar designs, our method could be readily adapted to other popular deep learning frameworks. By implementing our specialized toolkit—comprising MEFT, EMT, and PFT—in libraries such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016), Caffe (Jia et al., 2014), and MXNet (Chen et al., 2015), MLM4DNN can be transferred to these frameworks. Moreover, MLM4DNN can be flexibly extended by 1) summarizing more element types to cover new bugs and 2) reusing our workflow to construct dataset encompassing a wider range of DNN architectures, thereby adapting our approach to new bugs or architectures.

Internal Threat. To construct $Benchmark_{APR4DNN}$ containing single-hunk bugs, based on buggy models from DeepFD (Cao et al., 2022), we decomposed each bug location and the corresponding repair action into an independent repair task, involving manual inspection. To reduce the threat, all models are double-checked by two authors and publicly accessible at our repository [9]. In the future, we plan to expand our research to support the repair of multi-hunk bugs.

6 Conclusion

In this work, we propose MLM4DNN, a novel element-based DNN repair approach powered by a fine-tuned MLM, generating patches by predicting masked elements within the buggy DNNs. Based on nine element types defined for the DNN repair task, we construct a large-scale element prediction dataset from the latest commits of top-rated GitHub repositories. We then fine-tune an MLM to learn the element prediction tasks and design an element-based method to transform DNN repair tasks to element prediction tasks. Next, we develop a post-processing tool to filter out low-quality patches by leveraging light-weight static analysis. Finally, a validation tool is utilized to validate the generated patches. Our extensive experiments demonstrate that MLM4DNN significantly outperforms six baselines across three metrics, establishing its effectiveness in improving DNN repair performance, particularly when leveraging a fine-tuned UniXcoder. Additionally, it surpasses methods using the same LLM alone, confirming the effectiveness of our element-based DNN repair method.

7 Data Availability

Our source code and experimental data have been made publicly available for access and reuse, which can be found at **https://github.com/PGZXB/MLM4DNN**.

Acknowledgments

This work was supported partly by National Key Research and Development Program of China (No.2022YFB4502003), partly by National Natural Science Foundation of China (No. 62072017), Engineering Research Center of Intelligent Computing for Complex Energy Systems, Ministry of Education (No. ESIC202201) and State Key Laboratory of Complex & Critical Software Environment.

[9] https://github.com/PGZXB/MLM4DNN

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: Automated Fault Diagnosis and Localization for Deep Learning Programs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 573–585. doi:10.1145/3510003.3510099

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959. doi:10.1109/TSE.2019.2940179

Francois Chollet. 2015. Keras: Deep Learning for humans. https://keras.io. [Accessed 18-03-2024].

Jacob Cohen. 1992. A power primer. *Psychological bulletin* 112, 1 (1992), 155.

Aminu Da'u and Naomie Salim. 2020. Recommendation system based on deep learning methods: a systematic review and new directions. *Artif. Intell. Rev.* 53, 4 (2020), 2709–2748. doi:10.1007/S10462-019-09744-1

Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (Lecture Notes in Computer Science, Vol. 11424)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.). Springer, 171–191. doi:10.1007/978-3-030-16722-6_10

Di Feng, Christian Haase-Schütz, Lars Rosenbaum, Heinz Hertlein, Claudius Gläser, Fabian Timm, Werner Wiesbeck, and Klaus Dietmayer. 2021. Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges. *IEEE Trans. Intell. Transp. Syst.* 22, 3 (2021), 1341–1360. doi:10.1109/TITS.2020.2972974

Zhangyin Feng, Daya Guo, Duyu Tang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=hQwb-lbM6EL

Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1147–1158. doi:10.1145/3377811.3380415

Ali Ghanbari. 2020. ObjSim: lightweight automatic patch prioritization via object similarity. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 541–544. doi:10.1145/3395363.3404362

Ali Ghanbari and Andrian Marcus. 2022. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 654–665. doi:10.1145/3533767.3534368

Ali Ghanbari, Deepak-George Thomas, Muhammad Arbab Arshad, and Hridesh Rajan. 2023. Mutation-based Fault Localization of Deep Neural Networks. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1301–1313. doi:10.1109/ASE56229.2023.00171

Sorin Mihai Grigorescu, Bogdan Trasnea, Tiberiu T. Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *J. Field Robotics* 37, 3 (2020), 362–386. doi:10.1002/ROB.21918

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. doi:10.18653/V1/2022.ACL-LONG.499

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 1345–1351. doi:10.1609/AAAI.V31I1.10742

Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1162–1174. doi:10.1109/ASE56229.2023.00181

Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1110–1121. doi:10.1145/3377811.3380395

Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: mutation testing of deep learning systems based on real faults. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 67–78. doi:10.1145/3460319.3464825

Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 510–520. doi:10.1145/3338906.3338955

Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: fix patterns and challenges. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1135–1146. doi:10.1145/3377811.3380378

Gunel Jahangirova and Paolo Tonella. 2020. An Empirical Evaluation of Mutation Operators for Deep Learning Systems. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 74–84. doi:10.1109/ICST46399.2020.00018

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu (Eds.). ACM, 675–678. doi:10.1145/2647868.2654889

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. doi:10.1109/ICSE48619.2023.00125

Hyeyoung Ko, Suyeon Lee, Yoonseo Park, and Anna Choi. 2022. A survey of recommendation systems: recommendation models, techniques, and application fields. *Electronics* 11, 1 (2022), 141.

Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*.

Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. doi:10.1007/S10664-019-09780-Z

Guochang Li, Chen Zhi, Jialiang Chen, Junxiao Han, and Shuiguang Deng. 2024. Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 719–731. doi:10.1145/3691620.3695066

Yu Li, Muxi Chen, and Qiang Xu. 2022a. HybridRepair: towards annotation-efficient repair for deep learning models. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 227–238. doi:10.1145/3533767.3534408

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022b. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 511–523. doi:10.1145/3510003.3510177

Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, Tianlu Wang, Shuohui Chen, Daniel Simig, Myle Ott, Naman Goyal, Shruti Bhosale, et al. 2021. Few-shot learning with multilingual language models. *arXiv preprint arXiv:2112.10668* (2021).

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. doi:10.1145/3293882.3330577

Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. doi:10.1145/3395363.3397369

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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 175–186. doi:10.1145/3236024.3236082

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving Fault Localization and Program Repair with Deep Semantic Features and Transferred Knowledge. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1169–1180. doi:10.1145/3510003.3510147

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1456–1468. doi:10.1109/ICSE48619.2023.00127

J. A. Nelder and R. W. M. Wedderburn. 1972. Generalized Linear Models. *Journal of the Royal Statistical Society. Series A (General)* 135, 3 (1972), 370–384. http://www.jstor.org/stable/2344614

Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2022. Automatic Fault Detection for Deep Learning Programs Using Graph Transformations. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2022), 14:1–14:27. doi:10.1145/3470006

OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt. [Accessed 24-03-2024].

OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o. [Accessed 24-08-2024].

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 69–75. doi:10.1145/3524459.3527351

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. https://jmlr.org/papers/v21/20-074.html

Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 13–24. doi:10.1109/ICSE.2019.00020

Eldon Schoop, Forrest Huang, and Bjoern Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, and Steven Mark Drucker (Eds.). ACM, 310:1–310:16. doi:10.1145/3411764.3445538

Richard Schumi and Jun Sun. 2023. Semantic-Based Neural Network Repair. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 150–162. doi:10.1145/3597926.3598045

Shashi Pal Singh, Ajai Kumar, Hemant Darbari, Lenali Singh, Anshika Rastogi, and Shikha Jain. 2017. Machine translation using deep learning: An overview. In *2017 International Conference on Computer, Communications and Electronics (Comptelix)*. 162–167. doi:10.1109/COMPTELIX.2017.8003957

Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2023. Arachne: Search-Based Repair of Deep Neural Networks. *ACM Trans. Softw. Eng. Methodol.* 32, 4 (2023), 85:1–85:26. doi:10.1145/3563210

Felix Stahlberg. 2020. Neural Machine Translation: A Review. *J. Artif. Intell. Res.* 69 (2020), 343–418. doi:10.1613/JAIR.1.12007

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

Muhammad Usman, Divya Gopinath, Youcheng Sun, Yannic Noller, and Corina S. Păsăreanu. 2021. NNrepair: Constraint-Based Repair of Neural Network Classifiers. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 3–25. doi:10.1007/978-3-030-81685-8_1

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18, 6 (2024), 186345. doi:10.1007/S11704-024-40231-1

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. doi:10.18653/V1/2021.EMNLP-MAIN.685

Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 561–572. doi:10.1145/3510003.3510071

Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 251–262. doi:10.1109/ICSE43902.2021.00034

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 172–184. doi:10.1145/3611643.3616271

Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 479–490. doi:10.1109/SANER.2019.8668043

Huanhuan Wu, Zheng Li, Zhanqi Cui, and Jianbin Liu. 2022. GenMuNN: A mutation-based approach to repair deep neural network models. *Int. J. Model. Simul. Sci. Comput.* 13, 2 (2022), 2341008:1–2341008:17. doi:10.1142/S1793962323410088

Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023a. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 522–534. doi:10.1109/ASE56229.2023.00047

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023b. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1482–1494. doi:10.1109/ICSE48619.2023.00129

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. doi:10.1145/3540250.3549101

Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 819–831. doi:10.1145/3650212.3680323

Minrui Xu, Hongyang Du, Dusit Niyato, Jiawen Kang, Zehui Xiong, Shiwen Mao, Zhu Han, Abbas Jamalipour, Dong In Kim, Xuemin Shen, et al. 2024. Unleashing the Power of Edge-Cloud Generative AI in Mobile Networks: A Survey of AIGC Services. *IEEE Commun. Surv. Tutorials* 26, 2 (2024), 1127–1170. doi:10.1109/COMST.2024.3353265

He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1506–1518. doi:10.1145/3510003.3510222

Hao Zhang and W. K. Chan. 2019. Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 376–387. doi:10.1109/ASE.2019.00043

Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Trans. Software Eng.* 48, 2 (2022), 1–36. doi:10.1109/TSE.2019.2962027

Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting Template-Based Automated Program Repair Via Mask Prediction. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 535–547. doi:10.1109/ASE56229.2023.00063

Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 359–371. doi:10.1109/ICSE43902.2021.00043

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. doi:10.1145/3468264.3468544

Received 2024-09-13; accepted 2025-01-14