Typestate-based Fault Localization of API Usage Violations in a Deep Learning Program

Fraol Batole*, Ruchira Manke *, Robert Dyer † Tien N. Nguyen ‡, and Hridesh Rajan*
* *Department of Computer Science, Tulane University*, New Orleans, Louisiana, USA*
{fbatole, rmanke, hrajan}@tulane.edu
† *University of Nebraska–Lincoln*, Lincoln, NE, USA, rdyer@unl.edu
‡ *Computer Science Department, The University of Texas at Dallas*, Dallas, TX, USA, tien.n.nguyen@utdallas.edu

***Abstract*—Deep Learning (DL) applications have become essential in numerous domains, yet they remain plagued by subtle bugs that cause 66% of crashes in production systems. These failures primarily stem from API usage violations in complex frameworks like TensorFlow, Keras, and PyTorch, where APIs lack formal specifications and interdependencies between operations remain undocumented. Traditional static analysis tools fail to address DL-specific constraints, such as data dependency between layers. To bridge this critical gap, we propose NEURALSTATE, an approach to detect performance and program crash bugs in a DL program. NEURALSTATE follows a four-step process: (i) gather specifications for Deep Learning operations from different sources; (ii) introduce abstract states to represent these Deep Learning operations; (iii) design formal rules for transitioning between states based on the specifications; (iv) utilize a combination of standard analysis techniques (i.e., typestate and value propagation) to identify bugs in a DL program. Our evaluation on real-world benchmarks demonstrates NEURALSTATE's effectiveness, achieving a 25% improvement in precision and 63% improvement in recall compared to state-of-the-art tools. Most importantly, NEURALSTATE successfully detects 18 subtle bugs in 45 real-world programs that existing techniques miss entirely.**

***Index Terms*—Fault localization, Deep learning, Program analysis**

I. Introduction

Deep Learning (DL) programs are increasingly popular [1], typically developed using APIs from libraries such as TensorFlow, Keras, and PyTorch, which offer predefined operations for designing and configuring neural networks. Despite their maturity, developers face persistent challenges—Keras-related questions account for 20-35% of all Stack Overflow activity in 2024 [2]. These challenges have significant implications; API misuse causes program crashes in 66% of cases, terminating model training and wasting computational resources [3]. Traditional static analysis tools like PyLint [4] cannot detect these structural flaws due to the complexities of DL APIs, leaving developers vulnerable to subtle but critical errors.

These bugs, detectable before training, manifest as (1) layer compatibility violations, where the sequence of layers violates DL architectural constraints (e.g., missing a Flatten layer before Dense), (2) API protocol violations where required function calls are missing or incorrectly ordered, or (3) parameter inconsistencies between interdependent layers [3]. Moreover, popular frameworks like TensorFlow, Keras, and PyTorch offer rich but complex APIs with numerous distinct operations, each with several configurable parameters. For instance, configuring the 'Conv2D' layer in Keras involves setting multiple parameters such as filter size, kernel size, strides, and padding type, all of which must be chosen appropriately to ensure correct model behavior [5]. However, the relationships between these operations often lack formal specifications, creating a significant cognitive burden for developers.

Given the complexity and frequency of these DL API bugs, automated detection approaches are essential. Traditional program analysis techniques, such as PyLint [4], fall short as they lack an understanding of DL-specific constraints, such as the correct ordering of operations and parameter compatibility, as well as the complex relationships between DL operations. For instance, while they can detect undefined variables or type mismatches, they cannot verify whether a Dense layer's activation function is compatible with the model's loss function. Additionally, they cannot determine whether necessary reshaping operations are performed before feeding convolutional features into dense layers.

Current State-of-the-Art in DL Bug Detection and their Limitations. To address these challenges, researchers have proposed several specialized techniques for detecting bugs in DL programs [6, 7, 8]. While some focus on runtime performance issues or tensor shape errors [9, 10], they require access to training data and model training procedures. Thus, these techniques detect bugs during training. Our work aligns with bug detection tools that can detect issues before training begins. In this category, approaches can be broadly categorized into formal analysis-based techniques and LLM-based techniques. A more recent direction leverages Large Language Models (LLMs), such as LLMAPIDet [11], which uses an LLM to build a knowledge base of API misuse rules from code commits and applies few-shot prompting to generate patches. While powerful, LLM-based approaches rely on probabilistic knowledge of pre-trained models, which can be non-deterministic and act as black boxes. Among formal analysis approaches, NeuraLint [8] represents the state-of-the-art. NeuraLint models DL programs as graphs and defines verification rules to detect violations of common DL programming patterns. Using graph transformation techniques, it analyzes these program graphs against its rules to identify potential faults. However, NeuraLint's graph-based analysis has several limitations, primarily overlooking three crucial aspects of DL programs. These include the inability to resolve data dependencies, track interdependent configurations, and accurately construct control flow for higher-order functions. First, NeuraLint treats DL programs as sequential chains of operations, failing to resolve data dependencies between layers. This fundamental limitation in tracking data flow becomes particularly problematic in programs with multiple inputs, where different operations process different inputs through parallel branches of the network. Second, when one part of a DL program is modified (e.g., changing an activation function), NeuraLint cannot identify other parts that must be updated correspondingly (e.g., the matching loss function), as it lacks mechanisms to track these interdependent configurations. Third, DL programs often utilize higher-order functions where layers and their configurations are passed as parameters, making it challenging to construct accurate control flow from an abstract syntax tree (AST).

To illustrate the challenges in analyzing DL programs, consider the program from Stack Overflow [12] shown in Figure 1(a). This example exhibits multiple characteristics. First, the program processes two distinct inputs (lines 2-3), creating a branching neural architecture. Figure 1(b) shows how NeuraLint's analysis flattens this structure into a sequential chain of operations, losing critical information about data flow between layers. In contrast, the correct representation should preserve the branching structure as in Figure 1(c). Second, it contains an architectural bug where the final layer uses an incorrect activation function ('sigmoid' at line 23). This bug is particularly subtle as its fix requires modifying interdependent statements (i.e., changing the activation function to 'softmax' and updating the corresponding loss function to 'categorical_crossentropy'). Such interdependent configurations, which we refer to as co-changing statements, are common in DL programs but cannot be detected by approaches that analyze operations in isolation. Third, DL programs often utilize higher-order functions where layers and their configurations are passed as parameters (e.g., passing activations like 'relu' or 'sigmoid' as arguments to layer constructors), making it difficult to construct accurate control flow.

NEURALSTATE: Our Approach for DL Bug Detection. Recognizing that deep learning (DL) APIs implicitly dictate a protocol—valid transitions between layer types, required calls before certain configurations, and allowed parameter combinations–we model these protocols as finite state automata (FSA) and adopt typestate analysis, a technique proven effective for verifying API protocols in other domains [13, 14, 15, 16, 17], yet previously unexplored in the DL setting. Applying typestate to DL introduces unique challenges, such as capturing data dependency in multi-branch networks and analyzing parameter co-dependencies across complex, often higher-order function calls. To address these challenges, NEURALSTATE combines typestate analysis with value propagation, thereby detecting both performance and program crash bugs by simultaneously verifying valid sequences of layer operations and ensuring parameter consistency. Specifically, we break down each DL operation (e.g., Conv2D, Dense, Flatten) into abstract states, creating a finite representation of permissible transitions and constraints. Beyond operation ordering, we propagate values (such as activation and loss function types) to uncover incompatibilities across different layers. Finally, NEURALSTATE constructs a data-dependency-sensitive graph that preserves the branching input structures, and higher-order control flows unique to DL code, providing a more accurate view of real-world program behavior than the flattened representations used in prior work.

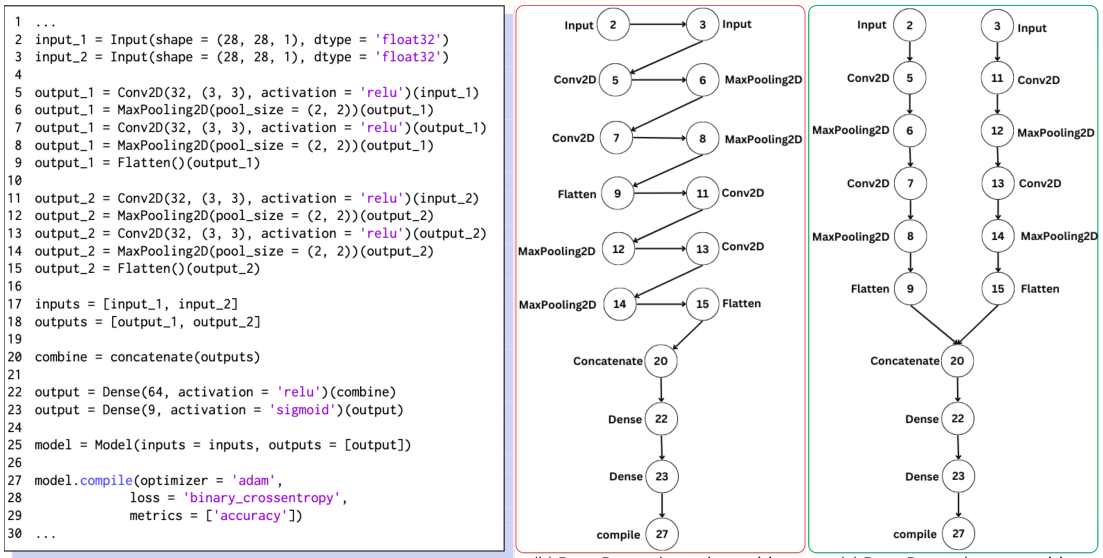
Key Technical Components. Our approach, NEURALSTATE consists of three main components:

1) *Capturing Data Dependency.* We leverage Keras's layer inspection API to extract the program's DL architecture. Then, we abstract the framework-specific layer information into an analyzable form called a NeuralState Sequence (NSS) using our defined grammar (Section II-B).
2) *Co-changing Statement Analysis.* While co-changing patterns are well-studied in traditional software maintenance [18], we identify a previously unrecognized class of such dependencies in DL programs that manifest between operation parameters. A common pattern we discover is the coupling between activation functions and loss functions, where modifying one requires corresponding updates to the other. To handle these interdependencies, we introduce context-sensitive analysis that tracks operation inputs across dependent layers.
3) *Unified Protocol Verification.* Detecting DL bugs requires analyzing both operation sequences and their configurations. We encode DL specifications as a finite state automaton with state transitions for DL operations. These specifications are then verified using a combination of typestate analysis for operation ordering and value propagation for parameter consistency. This integration enables verification of both architectural correctness (e.g., required layer orderings) and valid parameters (e.g., compatible activation/loss function).

We evaluate NEURALSTATE and NeuraLint on **two real-world benchmarks**: (1) NLBench, collected by NeuraLint [8], and (2) ExternalBench, curated by Humbatova et al. [19]. On NLBench, NeuralState achieves a 35.1% improvement in precision and 19.4% relative gain in recall compared to NeuraLint. The improvements are even more significant on ExternalBench, where NeuralState shows a 15.5% relative improvement in precision and 107% improvement in recall. These results across independently curated benchmarks demonstrate NeuralState's effectiveness in detecting DL bugs. An ablation study further validates that the combination of typestate analysis and value propagation is integral to NeuralState's superior performance. Detailed experimental outcomes and comparative analyses are presented in Section VI.

This work makes the following key contributions:

1) We introduce NEURALSTATE, a novel approach in the DL domain that integrates typestate analysis with value propagation to detect usage protocol violations.
2) We propose a representation of DL programs that accounts for branching architectures, multiple inputs, and higher-order function calls.
3) We design abstract states for DL operations, along with formal state transition rules that encode valid usage sequences and parameter constraints.
4) We evaluate NEURALSTATE against state-of-the-art static



```
1  ...
2  input_1 = Input(shape = (28, 28, 1), dtype = 'float32')
3  input_2 = Input(shape = (28, 28, 1), dtype = 'float32')
4
5  output_1 = Conv2D(32, (3, 3), activation = 'relu')(input_1)
6  output_1 = MaxPooling2D(pool_size = (2, 2))(output_1)
7  output_1 = Conv2D(32, (3, 3), activation = 'relu')(output_1)
8  output_1 = MaxPooling2D(pool_size = (2, 2))(output_1)
9  output_1 = Flatten()(output_1)
10
11 output_2 = Conv2D(32, (3, 3), activation = 'relu')(input_2)
12 output_2 = MaxPooling2D(pool_size = (2, 2))(output_2)
13 output_2 = Conv2D(32, (3, 3), activation = 'relu')(output_2)
14 output_2 = MaxPooling2D(pool_size = (2, 2))(output_2)
15 output_2 = Flatten()(output_2)
16
17 inputs = [input_1, input_2]
18 outputs = [output_1, output_2]
19
20 combine = concatenate(outputs)
21
22 output = Dense(64, activation = 'relu')(combine)
23 output = Dense(9, activation = 'sigmoid')(output)
24
25 model = Model(inputs = inputs, outputs = [output])
26
27 model.compile(optimizer = 'adam',
28               loss = 'binary_crossentropy',
29               metrics = ['accuracy'])
30 ...
```

(a) An Example of a Usage Protocol Violation

(b) Data-Dependence-insensitive Control Flow

(c) Data-Dependence-sensitive Control Flow

Note: The numbers in the circles represent the line number of an operation.

Fig. 1: Control flow representations: (a) DL program with multiple inputs, (b) NeuraLint's sequential representation missing data dependencies, and (c) NEURALSTATE's data-dependency-sensitive representation

analysis tool on two independently curated DL bug benchmarks. Our results show substantial improvements in both precision and recall, affirming the effectiveness of our approach in detecting real-world DL API misuse.

II. PRELIMINARIES

In this section, we outline the DL bugs supported by NEURALSTATE and formalize the components necessary for our analysis. First, we present a grammar for representing common DL operations. Then, we describe our process for collecting DL specifications and encoding them as finite state automata. Finally, we introduce our typestate-based approach for modeling DL program behavior through abstract states.

TABLE I: Grammar Representing Supported DL Operations

Symbol	Definition	Description
N	::= $L :: N \mid L$	Model composed of one or more layers
L	::= `Input(...)`	Input layer
	| `Conv2D(v, k, `a_f`, ...)`	2D convolutional layer
	| `Conv1D(v, k, `a_f`, ...)`	1D convolutional layer
	| `MaxPooling2D(k)`	2D max pooling layer
	| `MaxPooling1D(k)`	1D max pooling layer
	| `Flatten()`	Layer that flattens input
	| `Dense(v, `a_f`, ...)`	Fully connected layer
	| `Dropout(r)`	Dropout layer
	| `LayerNormalization()`	Normalize layer
	| `BatchNormalization()`	Batch Normalize layers
	| `Concatenate(`$L :: L$`)`	Merge multiple layers
	| `Compile(`l_f`, o, ...)`	Compile the neural network
a_f	::= $linear$	Activation function
	| $relu$	Activation function
	| $softmax$	Activation function
	| $sigmoid$	Activation function
	| $tanh$	Activation function
l_f	::= $binary\_crossentropy$	Loss function of the model
	| $categorical\_crossentropy$	
o	::= $adam \mid sgd$	Optimizer function of the model
v	::= $x \mid x \in \mathbb{Z}^+$	Number of units or neurons
k	::= $(x, y) \mid x \in \mathbb{Z}^+, y \in \mathbb{Z}^+$	kernel or pooling size
r	::= $x \mid x \in \mathbb{R}^+$	Dropout rate

A. *Supported DL Bugs*

NEURALSTATE aims to identify a wide range of bugs that commonly occur in DL programs before training starts. Categorizing these bugs is essential for systematically understanding their impact and developing targeted solutions, which ensures comprehensive detection and resolution of faults. The categorization adopted in this work is based on prior research on DL bug characteristics, providing a structured and empirically validated approach to bug identification [8]. The categories are outlined below:

- **Incorrect Model Parameter or Structure (IPS):** These bugs relate to errors in defining the model architecture, such as incorrect weight or bias initialization, missing activation functions, or improperly configured activation layers. Such issues often lead to training instability, high loss, and low accuracy, significantly reducing the model's effectiveness.
- **Tensor Shape Incompatibility (TSI):** DL programs rely on operations involving tensors (multi-dimensional arrays) that require shape compatibility for successful execution. Tensor shape mismatches can disrupt the flow of data through the network, causing runtime errors or inefficient memory usage. Ensuring tensor shapes are aligned is critical for maintaining the correct propagation of activations and gradients, ultimately allowing the model to train and infer accurately. Bugs in this category typically arise from missing reshape layers or mismatched tensor dimensions, leading to runtime errors or performance issues.
- **API Misuse (APIM):** API misuse refers to incorrect configuration choices within the DL framework, such as selecting the wrong optimizer or an invalid loss function. Such improper selections can lead to unstable training dynamics, ultimately hindering model convergence.
- **Structure Inefficiency (SI):** Structural inefficiencies in DL programs often result from improper architectural

configurations. These can include suboptimal filter sizes, kernel dimensions, or stride lengths in convolutional layers, as well as incorrect placement of dropout or batch normalization layers, or inappropriate dropout rates.

B. Supported DL Operations

We denote deep neural network as N in our grammar, where N is composed of layers (L) and ('::') denotes the concatenation of layers. These layers are fundamental building blocks that transform input data into an output. As shown in Table I, our approach supports common deep-learning layers with real-world applications, such as image recognition and regression tasks. These common layers are selected following a related work by Nikanjam et al.[8].

C. Collecting DL Specifications

We collected DL specifications from prior studies [8], adding an extra rule from the DL library's official documentation (i.e., high dropout rate) [5]. In total, we encode 24 DL specifications using finite-state automaton. We have included all specifications as a supplement for further reading.

D. Finite-State Automaton Based Specification

After collecting the specifications, the next step is to encode them as a finite-state automaton (FSA). Our FSA is defined as a six-tuple: $(S, s_0, E, A, L, \delta)$, where S is the set of abstract states representing the different phases of a DL program's execution, s_0 is the initial state, E is the error state, A is the accepting state, L is the set of DL operations, and δ is the transition function.

The transition function $\delta : S \times L \rightarrow S$ takes the current state $s_c \in S$ and a DL operation $l \in L$ as input and determines the next state $s_d \in S$ based on the specification rules. If the transition is valid, the automaton moves to the new state s_d; otherwise, it transitions to the error state E, indicating a specification violation.

Through a comprehensive study of DL specifications and an analysis of common operations employed in DL models, as documented in prior work [8], we derived a set of 11 abstract states that encompass the phases and operations encountered in DL programs. Table II presents the abstract states with a brief description.

Our approach formalizes the DL specifications as a finite-state automaton (FSA) and abstracts the DL program states into a finite set of abstract states. This formalization enables systematic verification of DL programs against the ground truth specifications. Consequently, we leverage typestate analysis, which is particularly suitable for enforcing behavioral constraints and ensuring that an object transitions through a sequence of valid states defined by a formal specification or protocol [13, 20].

III. APPROACH

This section presents NEURALSTATE, an approach for detecting bugs in DL programs by combining typestate analysis and value propagation techniques.

A. Overview

Figure 2 illustrates the workflow of NEURALSTATE, which consists of three main steps:

- **Extracting Layer Information:** NEURALSTATE takes a DL program as input and begins by extracting the statements and identifying the data dependencies between them (1). This step is performed using the `.layers()` API provided by the Keras library, which allows NEURALSTATE to access layer information before the model training begins, eliminating the need for the training dataset or the actual training process.
- **Constructing the NeuralState Sequence (NSS):** The extracted data dependencies are used to construct a representation of the DL program called the NeuralState Sequence (2). The NSS serves as an abstract form of the DL model, capturing the data-dependence sensitive control flow, where nodes represent states and edges denote the executed operations.
- **Identifying DL Violations:** NEURALSTATE applies a combination of typestate analysis and value propagation techniques, collectively referred to as NeuralState Analysis (NSA), to identify DL violations (3). Typestate analysis is used to identify invalid sequences of operations or API calls. Value propagation, in contrast, inspects the specific arguments passed to layer parameters to detect invalid configurations. For example, a common bug pattern occurs when all neuron weights in a layer are initialized to the same constant value, which prevents the layer from learning effectively. Our analysis uses value propagation to check the `kernel_initializer` parameter. It ensures that the keyword argument `kernel_initializer` takes a random value as input (i.e., setting like `kernel_initializer=HeNormal()`) as it allows each neuron to learn independently. Conversely, incorrect input of the keyword argument, such as `kernel_initializer=Zeros()`, will be flagged as a violation.

The following sections detail how we realize each component of our approach: § IV presents our data-dependence-sensitive representation, § V explains our integrated verification approach, and § V-B describes our handling of co-changing statements.

IV. DL REPRESENTATION

This section introduces the NeuralState Sequence, a novel representation that captures a DL program's data-dependence-sensitive control flow. The NSS provides a formal abstraction of the DL model's structure and enables precise analysis of its behavior.

A. **Preliminary**

Definition 1. *(**NeuralState Sequence**) The NeuralState Sequence (NSS) is a recursive representation of a DL program's execution states. It consists of abstract state nodes, denoted by s, and layer operations, denoted by l, which represent transitions between these nodes.*

TABLE II: States Representing Phases in a Deep Learning Program

Abstract State	Description
Init	The initial state of the program.
Hidden	Represents the initial layers of the neural network, involving input data processing.
Dense	Represents dense or fully-connected layers in the neural network.
Conv	Represents convolutional layers, including both 1D and 2D convolutions.
Pooling	Represents pooling layers used in conjunction with convolutional layers.
Reshape	Represents reshaping operations, often required after convolutional or pooling layers to match the input shape for subsequent layers.
Regularize	Represents regularization techniques, such as dropout, to prevent overfitting.
Normalize	Represents normalization layers, such as batch normalization or layer normalization.
Merge	Represents operations that merge or concatenate multiple input tensors.
LastHidden	Represents the final dense layer before the output layer, responsible for transforming features into the desired output format.
Compiled	The accepting state, representing the final configuration of the neural network, including the loss function and optimizer.

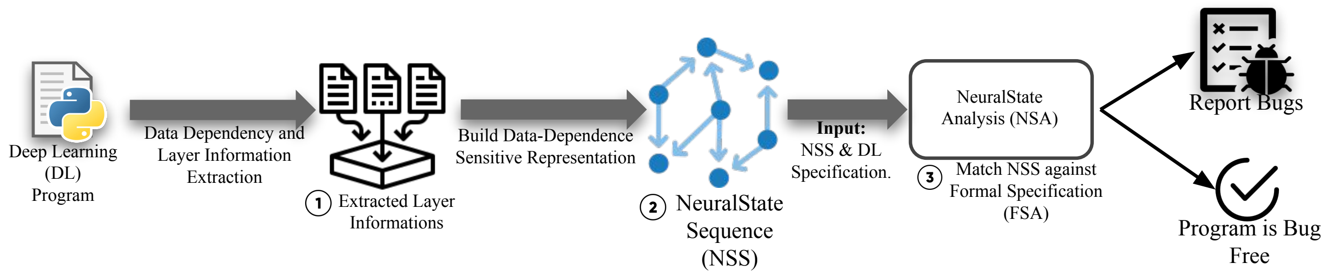
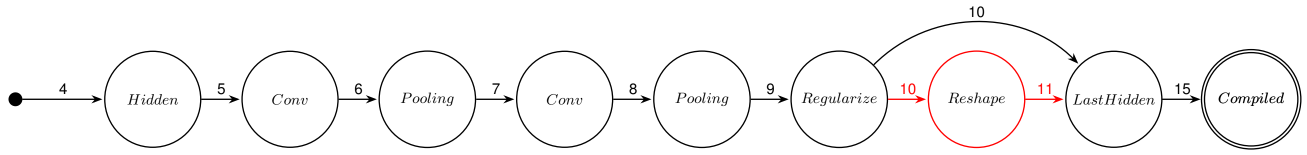

Deep Learning (DL) Program
Data Dependency and Layer Information Extraction
1 Extracted Layer Informations
Build Data-Dependence Sensitive Representation
2 NeuralState Sequence (NSS)
Input: NSS & DL Specification.
NeuralState Analysis (NSA)
3 Match NSS against Formal Specification (FSA)
Report Bugs
Program is Bug Free

Fig. 2: The workflow of NeuralState



NB. The numbers on the edge correspond to the layers of a DL model, following the line numbering in Fig 4. The nodes symbolize states.

Fig. 3: A NeuralState Sequence for the DL program shown in Figure 4, with missing state in red

If a layer operation l exists, it connects the current state s to the next state in the sequence, producing a concatenated sequence of states and operations, $s\ l$ :: NSS. When no further operations apply, the sequence terminates at a final state s, marking the completion of the control flow for that program segment. Here, :: denotes concatenation, linking each state transition within the sequence.

B. Constructing the NSS

Let P be a DL program defining a neural network N. We construct the NeuralState Sequence NSS(P) by analyzing the structure of N and the relationships between its layers. First, we extract the sequence of layers $\mathcal{L} = [l_1, l_2, \ldots, l_m]$ from N using the `model.layers` attribute provided by the DL library (e.g., Keras). This attribute allows us to access the layers of the model in the order they are defined in the program.

To construct the NSS, we iterate over the layers in $\mathcal{L}$ and perform the following steps for each layer l_i:

1) Determine the corresponding abstract state $s_i \in S$ based on the type and characteristics of l_i. We define a mapping function $F : L \rightarrow S$, that maps each layer type to its corresponding abstract state according to the DL specification. Formally, $s_i = F(l_i)$.
2) Let $D(l_i) \subseteq \{l_1, \ldots, l_{i-1}\}$ denote the set of layers that l_i directly depends on, i.e., the layers whose outputs are used as inputs to l_i. Check if all the layers in $D(l_i)$ have already been processed and their corresponding state-operation pairs have been added to the NSS. This ensures that the data dependencies of l_i are satisfied.
3) Based on the dependency analysis in step 2, append the state-operation pair (s_i, l_i) to the NSS. This indicates that the layer operation l_i is applicable in the abstract state s_i and transitions the model to the next state.

Example of NSS: Figure 3 illustrates the NSS for the DL program shown in Figure 4. Each node in the NSS represents an abstract state (e.g., Hidden, Conv, Pooling), and each edge represents a layer operation that transitions the program between states.

```
1  ...
2  model = keras.Sequential(
3      [
4          keras.Input(shape=(28, 28, 1)),
5          layers.Conv2D(32, kernel_size=(3, 3),
               activation="relu"),
6          layers.MaxPooling2D(pool_size=(2, 2)),
7          layers.Conv2D(64, kernel_size=(3, 3),
               activation="relu"),
8          layers.MaxPooling2D(pool_size=(2, 2)),
9          layers.Dropout(0.5),
10         + layers.Flatten()
11         layers.Dense(10, activation="softmax"),
12     ]
13 )
14 ...
15 model.compile(loss="categorical_crossentropy",
       optimizer="adam", metrics=["accuracy"])
16 model.fit(x_train, y_train,
       batch_size=batch_size, epochs=epochs,
       validation_split=0.1)
17 ...
```

```
1 NeuralState Error -> You need to flatten the
      layer before adding a Dense layer.
```

Fig. 4: A DL program with a crash bug and its error report

Analyzing the NSS reveals a missing Reshape state between the Regularize and LastHidden states, indicating a bug in the program's structure due to the omission of a required `Flatten()` layer at line 10. This insight, provided by NEURALSTATE, offers actionable feedback to guide developers in fixing the usage protocol violation.

V. DL PROTOCOL VIOLATION DETECTION

This section presents our approach to detecting DL protocol violations. We first present an algorithmic overview that provides an intuitive understanding of how NeuralState detects violations (§ V-A). We then complement this with a precise formulation of state transition rules that form the mathematical foundation of our typestate analysis approach (§ V-B). While these rules could be explained informally, a formal presentation eliminates ambiguity and ensures reproducibility, allowing other researchers to faithfully implement and build upon our work.

Algorithm 1 NSA Analysis

```
1:  procedure $NSA$(NSS, FSA)
2:      $\Gamma \leftarrow []$                          ▷ Initialize empty context
3:      $s_c \leftarrow NSS.getFirstNode()$               ▷ Starting in 'init' state
4:      $s_d \leftarrow NSS.getNextNode(s_c)$
5:      Violations $\leftarrow []$
6:      while $s_d \neq null$ do
7:          $l \leftarrow NSS.getOperation(s_d)$
8:          $s_n \leftarrow FSA.\delta(s_c, l)$     ▷ Determine the next state based on $\delta$
9:          if $s_n = E$ then                     ▷ Check if it is an invalid transition
10:             Violations $\leftarrow$ Violations $\cup \{(s_c, s_d)\}$
11:         else
12:             $\Gamma \leftarrow UpdateContext(\Gamma, s_c, l)$          ▷ Update context
13:         $s_c \leftarrow s_d$
14:         $s_d \leftarrow NSS.getNextNode(s_c)$
15:     return Violations, $\Gamma$
```

$$\frac{\Gamma \vdash \delta(s,l) = s', \Gamma' \quad \Gamma' \vdash NSS = s'', \Gamma'' \quad s'' \in A}{\Gamma \vdash s\ l :: NSS = A, \Gamma''} \quad \text{(NSA)}$$

$$\frac{\Gamma, \mathit{Hidden} \mapsto \mathit{True} = \Gamma'}{\Gamma \vdash \delta(\mathit{Init}, \texttt{Input()}) = \mathit{Hidden}, \Gamma'} \quad (R_1)$$

$$\frac{\begin{array}{c}\Gamma(\mathit{Conv}) = \mathit{False} \qquad \Gamma(\mathit{Reshape}) = \mathit{False} \\ v \in \mathbb{Z}^+ \quad a_f \in \{\mathit{relu}, \mathit{tanh}\} \quad s_c \in \{\mathit{Flatten}, \mathit{Dense}, \mathit{Hidden}\} \\ \Gamma, \mathit{Dense} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, }a_f\texttt{)}) = \mathit{Dense}, \Gamma'} \quad (R_2)$$

$$\frac{\begin{array}{c}\Gamma(\mathit{Reshape}) = \mathit{True} \\ v \in \mathbb{Z}^+ \quad a_f \in \{\mathit{relu}, \mathit{tanh}\} \quad s_c \in \{\mathit{Flatten}, \mathit{Dense}, \mathit{Hidden}\} \\ \Gamma, \mathit{Dense} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, }a_f\texttt{)}) = \mathit{Dense}, \Gamma'} \quad (R_3)$$

$$\frac{\begin{array}{c}l \in \{\texttt{Conv1D(v, k, }a_f\texttt{)}, \texttt{Conv2D(v, k, }a_f\texttt{)}\} \\ v \in \mathbb{Z}^+ \quad k \in \mathbb{Z}^+ \quad a_f = \mathit{relu} \quad s_c \in \{\mathit{Hidden}, \mathit{Pooling}\} \\ \Gamma, \mathit{Conv} \mapsto \mathit{True}, c_v \mapsto v = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, l) = \mathit{Conv}, \Gamma'} \quad (R_4)$$

$$\frac{\begin{array}{c}l \in \{\texttt{Conv1D(v, k, }a_f\texttt{)}, \texttt{Conv2D(v, k, }a_f\texttt{)}\} \\ v \in \mathbb{Z}^+ \quad k \in \mathbb{Z}^+ \quad a_f = \mathit{relu} \quad s_c \in \{\mathit{Hidden}, \mathit{Pooling}\} \\ \Gamma(\mathit{Conv}) = \mathit{True} \qquad \Gamma(c_v) \leq v \qquad \Gamma, c_v \mapsto v = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, l) = \mathit{Conv}, \Gamma'} \quad (R_5)$$

$$\frac{\begin{array}{c}l \in \{\texttt{MaxPooling1D(k)}, \texttt{MaxPooling2D(k)}\} \\ k \in \mathbb{Z}^+ \qquad \Gamma, \mathit{Pooling} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(\mathit{Conv}, l) = \mathit{Pooling}, \Gamma'} \quad (R_6)$$

$$\frac{\begin{array}{c}\Gamma(\mathit{Conv}) = \mathit{True} \quad s_c \in \{\mathit{Conv}, \mathit{Pooling}\} \\ \Gamma, \mathit{Reshape} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Flatten()}) = \mathit{Reshape}, \Gamma'} \quad (R_7)$$

$$\frac{\begin{array}{c}l \in \{\texttt{layerNormalization()}, \texttt{batchNormalization()}\} \\ s_c \in \{\mathit{Dense}, \mathit{Conv}, \mathit{Pooling}\} \quad \Gamma, \mathit{Normalize} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, l) = \mathit{Normalize}, \Gamma'} \quad (R_8)$$

$$\frac{\begin{array}{c}0.8 \geq r \geq 0 \quad s_c \in \{\mathit{Dense}, \mathit{Conv}, \mathit{Pooling}\} \\ \Gamma, \mathit{Regularize} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dropout(r)}) = \mathit{Regularize}, \Gamma'} \quad (R_9)$$

$$\frac{s_c \in \{\mathit{Dense}, \mathit{Conv}, \mathit{Flatten}\} \qquad \Gamma, \mathit{Merge} \mapsto \mathit{True} = \Gamma'}{\Gamma \vdash \delta(s_c, \texttt{Concatenate(}L :: L\texttt{)}) = \mathit{Merge}, \Gamma'} \quad (R_{10})$$

$$\frac{\begin{array}{c}\Gamma(\mathit{Conv}) = \mathit{False} \qquad \Gamma(\mathit{Reshape}) = \mathit{False} \\ v \in \mathbb{Z}^+ \qquad a_f \in \{\mathit{linear}, \mathit{sigmoid}, \mathit{softmax}\} \\ s_c \in \{\mathit{Reshape}, \mathit{Regularize}, \mathit{Normalize}\} \\ \Gamma, f \mapsto a_f = \Gamma' \qquad \Gamma', \mathit{LastHidden} \mapsto \mathit{True} = \Gamma''\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, }a_f\texttt{)}) = \mathit{LastHidden}, \Gamma''} \quad (R_{11})$$

$$\frac{\begin{array}{c}\Gamma(\mathit{Reshape}) = \mathit{True} \\ v \in \mathbb{Z}^+ \qquad a_f \in \{\mathit{linear}, \mathit{sigmoid}, \mathit{softmax}\} \\ s_c \in \{\mathit{Reshape}, \mathit{Regularize}, \mathit{Normalize}\} \\ \Gamma, f \mapsto a_f = \Gamma' \qquad \Gamma', \mathit{LastHidden} \mapsto \mathit{True} = \Gamma''\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, }a_f\texttt{)}) = \mathit{LastHidden}, \Gamma''} \quad (R_{12})$$

$$\frac{\begin{array}{c}\Gamma(f) \in \{\mathit{linear}, \mathit{sigmoid}\} \qquad \Gamma \vdash l_f = \mathit{binary\_crossentropy} \\ o \in \{\mathit{adam}, \mathit{sgd}\} \qquad \Gamma, \mathit{Compiled} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(\mathit{LastHidden}, \texttt{Compile(}l_f\texttt{, o)}) = \mathit{Compiled}, \Gamma'} \quad (R_{13})$$

$$\frac{\begin{array}{c}\Gamma(f) = \mathit{softmax} \qquad \Gamma \vdash l_f = \mathit{categorical\_crossentropy} \\ o \in \{\mathit{adam}, \mathit{sgd}\} \qquad \Gamma, \mathit{Compiled} \mapsto \mathit{True} = \Gamma'\end{array}}{\Gamma \vdash \delta(\mathit{LastHidden}, \texttt{Compile(}l_f\texttt{, o)}) = \mathit{Compiled}, \Gamma'} \quad (R_{14})$$

Fig. 5: State Transition Rules for DL Specifications

TABLE III: Summary of Encoded Typestate Rules for Verifying DL API Usage

Rule ID	Description
R1	An `Input` layer must begin the model definition.
R2-R3, R11-R12	Permits `Dense` layers for hidden representations and verifies the final `Dense` layer has a valid activation for the task (e.g., softmax, sigmoid).
R4-R5	Allows a `Conv` layer to be added after the hidden layer or after another feature-extracting layer (e.g., `Pooling`).
R6	Enforces that a `Pooling` layer must follow a `Conv` layer to downsample feature maps.
R7	Requires a `Flatten` operation to transition from multi-dimensional convolutional features to a 1D vector for `Dense` layers.
R8	Allows Normalization layers (e.g., `BatchNormalization`) to be placed after convolutional or dense layers.
R9	Permits a `Dropout` layer after convolutional or dense layers to mitigate overfitting.
R10	Allows `Concatenate` to merge multiple parallel branches, a key step for our data-dependency tracking.
R13-R14	A co-change rule that verifies the loss function in `model.compile()` is compatible with the activation used in the final output layer.

A. NeuralState Analysis Algorithm

The NSA algorithm initializes an empty context (Γ) to track the program's execution history (line 2). It then sets the current state (s_c) to the initial state of the NSS and fetches the next transition state (s_d) from the current state (lines 3-4).

The algorithm iterates through the transitions in the NSS until there are no more transitions (lines 6- 14). For each transition, it retrieves the corresponding DL operation (l) from the NSS and determines the next state (s_n) using the FSA's δ.

If the next state (s_n) is the error state (E), indicating an invalid transition according to the state transition rules, the algorithm adds the transition pair (s_c, s_d) to the set of violations. Otherwise, it updates the context (Γ) based on the state transition rules using the $UpdateContext$ function (line 12). After processing each transition, the algorithm updates the current state (s_c) to the next transition state (s_d) and fetches the subsequent transition state from the NSS. Finally, after iterating through all transitions, the algorithm returns the set of violations and the final context (Γ).

B. Precise Formulation of State Transition Rules for DL

Figure 5 depicts the state transition rules, which utilize a context (Γ) to track the program's execution history. This context stores information about previously visited states and relevant values, enabling the analysis to handle state dependencies and co-changing statements. Table III provides short descriptions for these rules. Other rules can be defined in a similar manner.

The NSA rule serves as the entry point, initializing an empty context and inductively applying the transition function δ to validate each operation in the DL program's execution trace. If a transition is valid according to the defined rules, the current state is updated, and the analysis proceeds to the next operation. Otherwise, a violation is reported, indicating a potential bug in the program. For instance, the R_3 rule enforces constraints on the activation function and the number of units to a dense layer. To illustrate how the rules are read, consider the second R_3: "If the current state s_c is Flatten, Dense, or Hidden, the number of units v is a positive integer, the activation function a_f is non-linear (i.e., relu or tanh) and both Conv and Reshape has been visited before (based on the context Γ), then the transition to the Dense state is valid, and the context is updated to reflect that Dense has been visited."

Handling Co-Changing Layers. We handle these co-changing statements in the rules R_{11-14}. The R_{11-12} rule validates the final dense layer's activation function and the number of units based on the problem type (binary classification, multi-class classification, or regression). It then records the activation function used during that execution by updating the context (i.e., $f{\mapsto}a_f{=}\Gamma'$). In the R_{13-14} rule, the analysis consults the recorded activation function f to verify if it matches the proper loss function based on the specification.

Most importantly, if a violation is detected in the LastHidden state, indicating a potential inconsistency between the activation function and the number of units, the analysis assumes that a fix has occurred. This is a reasonable assumption because, in the final dense layer, the activation function is typically either a single-class or multi-class activation function, each with a specific loss function (binary_crossentropy or categorical_crossentropy, respectively). Therefore, the analysis negates the recorded activation function f and matches it against the corresponding loss function. One of the challenges in creating our specifications, particularly for co-changing statements, is that these critical dependencies are often implicit and undocumented in the official API guides. For instance, while the coupling between a `softmax` activation and a `categorical_crossentropy` loss function is a well-known pattern for multi-class classification, this rule is not formally enforced by the Keras API. Thus, developers learn it from experience and community examples []. Deriving such rules requires us to infer the developer's intent (e.g., binary vs. multi-class classification) from architectural clues, such as the number of neurons in the final `Dense` layer (e.g., `Dense(1)` for binary tasks versus `Dense(N)` for N-class tasks). This manual process of identifying tightly-coupled parameters among a vast combinatorial space introduces a bias towards the most common patterns.

One of the challenges in creating our specifications, particularly for co-changing statements, is that these critical dependencies are often implicit and undocumented in the official API guides. For instance, while the coupling between a `softmax` activation and a `categorical_crossentropy` loss function is a well-known pattern for multi-class classification, this rule is not formally enforced by the Keras API. Thus, developers learn it from experience and community examples. Deriving such rules requires us to infer the developer's intent (e.g., binary vs. multi-class classification) from architectural clues, such as the number of neurons in the final `Dense` layer (e.g., `Dense(1)` for binary tasks versus `Dense(N)` for N-class tasks). A promising direction for future work could be to investigate automated techniques to mine these implicit specifications from large-scale code corpora.

Finally, the `Compiled` rule represents the accepting state, verifying that the loss function and optimizer are consistent with the activation function used in the final layer.

VI. Empirical Evaluation

In this section, we describe the evaluation of our approach. First, we briefly present the research questions. Next, we describe our experimental methodologies.

A. Research Questions

Our evaluation aims to answer the following questions:

- **RQ1: Effectiveness Evaluation.**
  (A) How effective is NeuralState compared with the state-of-the-art NeuraLint on their own benchmark?
  (B) How effective is NeuralState compared with the state-of-the-art NeuraLint on an unseen benchmark?
- **RQ2: Comparative Analysis of Bug Detection Capabilities.** To what extent does NeuralState's bug detection overlap with NeuraLint's, and what unique bug detection strengths do NeuralState exhibit?
- **RQ3: Impact of Analysis Techniques.** How do the two key analysis techniques (value-propagation and typestate analysis) in NeuralState impact its effectiveness?
- **RQ4: Time Complexity Comparison.** How is the time-complexity of NeuralState compared to NeuraLint?

B. Experimental Methodology

In this section, we present the two real-world benchmarks used for evaluation, the evaluation metric, baselines, and implementation.

1) **Benchmarks:** To ensure a fair and informative comparison, we evaluate the performance of NeuralState on **two** benchmarks. The first benchmark, **NLBench**, is from NeuraLint's work [8] and contains 26 real-world Keras buggy programs collected from Stack Overflow (SO). The second benchmark, **ExternalBench**, is taken from a prior study by Humbatova et al. [19] and consists of 19 real-world buggy programs with complete Keras code collected from SO posts. In total, the benchmarks contain 22 bugs with a program crash symptom, 21 with bad performance, and 2 with incorrect functionality symptoms.

2) **Evaluation Metric:** We adopted the same evaluation metrics as in NeuraLint:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (1)$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (2)$$

We utilize the curve fitting method to evaluate the time complexity to obtain a model representing the data, as used in related works [14]. We used the coefficient of determination ($R^2 \in [0,1]$) to evaluate the model's effectiveness in fitting the data points. The closer the R^2 value is to 1, the more scalable the approach is.

3) **Baselines:** We compared our approach against the state-of-the-art DL bug detection tool, NeuraLint [8]. NeuraLint builds a graph representation of DL programs and runs a graph-based verification tool. It utilizes pre-defined rules to identify faults and design inefficiencies in these programs. Each rule is associated with a set of guidelines for resolving the issue, which is provided to the user if the rule is violated.

4) **Implementation:** NeuralState is implemented in Python, leveraging a suite of established libraries and custom tools to ensure a comprehensive approach to DL bug detection. Keras is employed for deep learning operations, offering the necessary flexibility for layer inspection and the analysis of neural architectures. NetworkX is used to build the NSS, enabling effective representation of dependencies and relationships within DL models. Typestate analysis is implemented through custom state management utilities specifically developed to track the phases of DL program execution, modeled as finite state automata (FSA). Additionally, value propagation enhances typestate analysis by leveraging Python's built-in data structures (e.g., dictionaries and sets) to accurately track configuration parameters across multiple operations. Finally, layer inspection is conducted using Keras's internal APIs, providing access to detailed layer metadata, including parameters and configurations. This deep inspection enables a thorough analysis of complex, real-world DL models, ensuring that all components comply with established architectural standards. All the experiments were conducted on an Intel Core i9 with 64 GB of 2400 MHz DDR4 memory. We use the Python *time* library to measure the execution time of detecting bugs.

This section presents the results of our experiments.

C. **Effectiveness Evaluation (RQ1)**

To evaluate NeuralState's effectiveness in detecting DL bugs, we divide our analysis into two parts. First, we compare it with NeuraLint on their own benchmark to establish a direct comparison with the state-of-the-art (RQ1-A). Then, to assess generalizability, we evaluate both tools on an independent benchmark containing different types of DL bugs (RQ1-B).

TABLE IV: Precision and Recall Results

Approach	NLBench		ExternalBench	
	Precision	Recall	Precision	Recall
NeuraLint [8]	74.0%	62.0%	86.6%	32.5%
NeuralState (our approach)	**100.0%** (+20.6%)	**74.0%** (+12.0%)	**100.0%** (+13.4%)	**67.5%** (+35.0%)

1) **Comparison on NLBench (RQ1-A):** *Experimental Setup.* We initially assessed the effectiveness of NeuralState on the first benchmark, NLBench, which is also utilized by NeuralLint.

Detailed Analysis. The detailed results of NeuralState's performance on NLBench are presented in Figure 6. The results reveal several interesting patterns across different bug categories. For Incorrect Model Parameter or Structure (IPS) bugs, NeuralState detected 7 instances compared to NeuraLint's 5, representing a 40% improvement. This enhanced detection capability stems from NeuralState's value propagation technique, which enables it to track interdependent configurations across layers, particularly in cases where activation functions and loss functions must be compatible. In the Tensor Shape Incompatibility (TSI) category, NeuralState identified 6 bugs versus NeuraLint's 4, a 50% increase. This improvement can be attributed to NeuralState's data-dependence-sensitive representation, which maintains the branching structure of neural architectures and better tracks shape transformations across parallel paths. For API Misuse (APIM), NeuralState detected 7 instances compared to NeuraLint's 5, showing a 40% improvement. This category particularly highlights the effectiveness of our typestate analysis in capturing invalid sequences of API calls and ensuring proper ordering of operations. Interestingly, both tools identified the same number of Structure Inefficiency (SI) bugs (17), suggesting that these bugs, which often involve suboptimal architectural choices, are equally detectable through both graph-based and typestate-based approaches. The consistent performance across all categories, with improvements in three out of four, demonstrates that NeuralState's combined approach of typestate analysis and value propagation effectively addresses the limitations of purely graph-based analysis, particularly in scenarios requiring a deeper understanding of data dependencies and co-changing statements.

Overview of Results. Table IV presents a summary of the performance comparison between NeuralState and NeuraLint on NLBench. NeuralState achieves a precision of 100% and a recall of 74%, outperforming NeuraLint by 20.6% and 12%, respectively. These improvements translate to relative gains of 35.1% in precision and 19.4% in recall, highlighting NeuralState's bug detection capabilities.

2) **Comparison on ExternalBench (RQ1-B)***:* *Experimental Setup.* To evaluate NeuralState's generalization capability, we curated ExternalBench from established DL fault localization benchmarks [6, 10, 19, 21]. Like the original studies, the dataset contains programs sourced from Stack Overflow to capture real-world bugs. As NeuralState supports FCNNs and CNNs, we filtered for programs using these architectures. After removing duplicates and programs overlapping with NLBench, ExternalBench provides a distinct set of validation programs, enabling comprehensive evaluation of NeuralState's bug detection capabilities.

Detailed Analysis. Figure 6 presents the detailed results of NeuralState's performance in detecting usage protocol violations on ExternalBench. The results reveal a notably different detection pattern compared to NLBench, with more pronounced improvements in certain categories. In the IPS category, NeuralState detected 6 bugs while NeuraLint failed to identify any, highlighting a significant advancement in detecting structural and parameter-related issues. This significant difference can be attributed to NeuralState's context-sensitive analysis, which is particularly effective at identifying subtle interactions between model parameters that occur more frequently in ExternalBench's diverse benchmark. For TSI bugs, both tools identified only one instance, suggesting that shape-related issues were less prevalent in this benchmark. The APIM category shows another substantial improvement, with NeuralState detecting 6 cases compared to NeuraLint's 3, a 100% increase. This improvement stems from NeuralState's typestate analysis being particularly adept at identifying API protocol violations in more complex program flows, which are characteristic of the real-world applications in ExternalBench. In the SI category, NeuralState identified 11 inefficiencies compared to NeuraLint's 8, representing a 37.5% improvement. This enhanced detection of structural inefficiencies can be attributed to NeuralState's ability to track value propagation across layers, enabling it to identify suboptimal architectural choices that might be missed by graph-based analysis alone. The overall pattern of improvements across ExternalBench, particularly in IPS and APIM categories, validates NeuralState's effectiveness on independently curated datasets and demonstrates its robustness in handling diverse, real-world DL bugs.

Overview Results. As illustrated in Table IV, NeuralState achieved 100% precision and 67.5% recall on ExternalBench, representing a 13.4% and 35% improvement over NeuraLint, respectively. This translates to a 15.5% relative improvement in precision and a notable 107% relative improvement in recall compared to NeuraLint.

TABLE V: True Positive Overlapping Analysis

Category	NLBench	ExternalBench	Total
Unique to NeuraLint	0	0	0
Overlap	30	12	42
Unique to NeuralState	**6**	**12**	**18**

D. Comparative Analysis of Bug Detection (RQ2)

Experimental Setup. We conducted a comparative analysis to examine the bugs detected by both NeuralState and NeuraLint, as well as the bugs uniquely identified by each tool. This analysis aimed to understand the strengths and limitations of the two approaches in detecting deep learning code bugs.

Results. Table V presents the results of the comparative analysis. NeuralState detected 18 unique bugs, demonstrating its superior bug detection capabilities. In contrast,

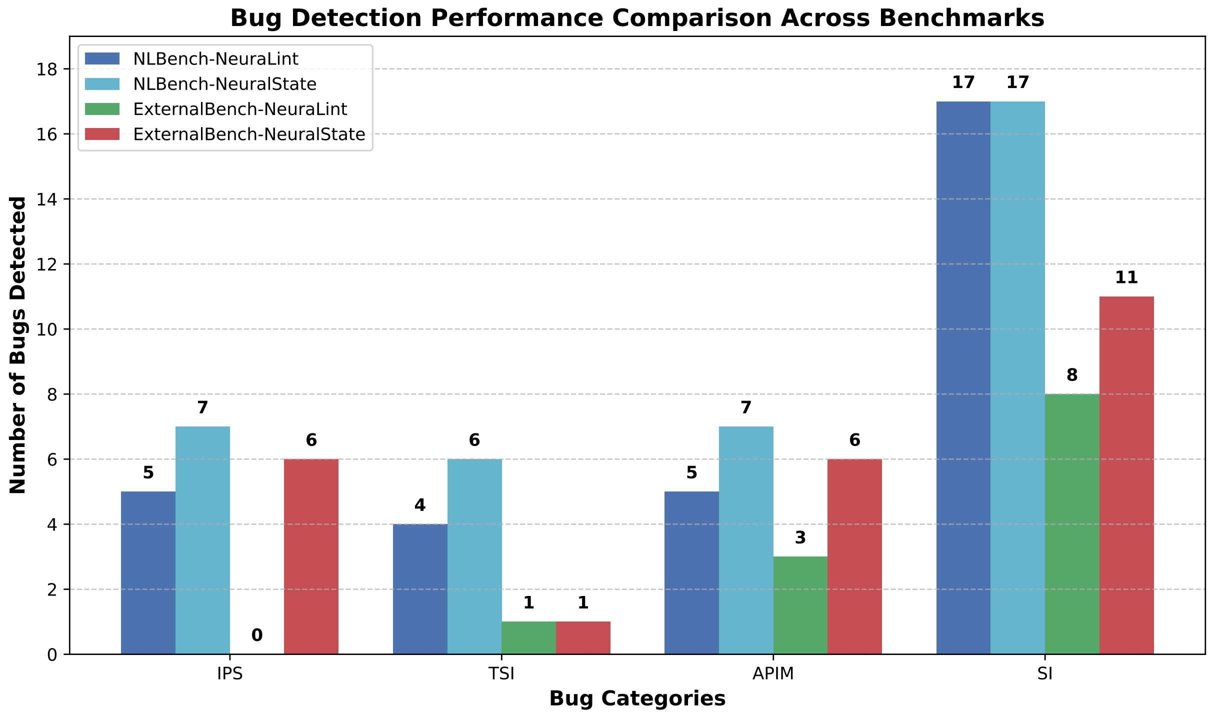


Fig. 6: Effectiveness of various approaches in detecting bugs across four bug sub-categories

NeuraLint did not detect any unique bugs beyond those found by NEURALSTATE, highlighting its limitations in handling data dependency and co-changing features. The table further reveals that both tools overlapped in detecting 42 bugs across the two benchmarks, indicating a common set of bugs that both approaches could effectively identify.

The discrepancy in bug detection capabilities, where NEURALSTATE identified 18 unique bugs, can be attributed to two key factors. First, NeuraLint's inability to identify co-changed statements hindered its ability to detect bugs that need multiple fixes. Second, out of the 18 cases where NEURALSTATE uniquely reported bugs, NeuraLint exhibited false positives or false negatives in 3 cases due to unresolved data dependencies.

```
1 ...
2 model = Sequential()
3 model.add(Conv1D(filters=20,
      kernel_size=4,activation='relu',
      padding='same', input_shape=(600,1)))
4 model.add(MaxPooling1D(pool_size = 2))
5 model.add(Dropout(0.3))
6 model.add(Flatten())
7 model.add(Dense(50, activation='relu', input_dim
      = 600))
8 model.add(Dense(1, activation='softmax'))
9 model.compile(loss="binary_crossentropy",
      optimizer="nadam", metrics=['accuracy'])
```

Fig. 7: NeuraLint false positive report from NLBench

To illustrate the limitations, we examine concrete examples:

1) **Case study with a buggy DL program (Program Crash):** Figure 7 shows a buggy program from the NL-BENCH [8] benchmark. It uses the sequential API to define 6 layers of convolutional and dense operations. During execution, the program crashes due to incompatibility between activation and loss function. To fix the bug, a developer should change 'softmax' to 'sigmoid' (line 8). Both tools accurately identify this bug. However, NeuraLint reports an additional false positive. It incorrectly identifies an issue with the loss function `'binary_crossentropy'` on line 9 since it does not consider the relation with the activation function in the statement on line 8. This highlights the significance of dependencies and co-changes in DL bug detection.

2) **Evaluation of the motivating example.:** Here, we evaluated NEURALSTATE and NeuraLint using the example in Section 1. NEURALSTATE correctly identifies two bugs at line 23 ('sigmoid' $\rightarrow$ 'softmax') and on line 28 ('binary_crossentropy' $\rightarrow$ 'categorical'). Additionally, NEURALSTATE did not report any false positives. NeuraLint reports 1 false positive and 0 true positives because it does not consider the data dependencies among the layers and considers that all hidden layers are applied to the second input. Our results validate two key observations: (1) resolving data dependencies and (2) identifying co-changed statements is important for effective bug detection in a DL program.

E. Impact of Analysis Techniques (RQ3)

Experimental Setup. To validate our third observation (Section I) and investigate the individual contributions of value propagation (VP) and typestate analysis (TSA) to NEURALSTATE's effectiveness, we conducted an ablation study with two variants: (1) NEURALSTATE without TSA, and (2) NEURALSTATE without VP. Since VP incorporates co-change analysis, the second variant's results help validate its impact on our approach. The dependence-sensitive analysis is core to NEURALSTATE, and removing it would reduce NEURALSTATE to the baseline NeuraLint.

TABLE VI: Impact of Analysis Techniques on NeuralState

	NLBENCH		**EXTERNALBENCH**	
Approach	**Precision**	**Recall**	**Precision**	**Recall**
NeuralState	**100.0%**	**74.0%**	**100.0%**	**67.5%**
- w/o TSA	**100.0%**	38.7%	**100.0%**	42.1%
- w/o VP	83.7%	42.8%	84.0%	31.5%

NB. TSA stands for typeState analysis, and VP stands for value propagation.

Results. Table VI presents the results of the ablation study, comparing the performance of NEURALSTATE with its two variants on both benchmarks, NLBENCH and EXTERNALBENCH. The full NEURALSTATE implementation, incorporating both value propagation and typestate analysis, achieved the highest precision and recall across both benchmarks.

When evaluating the variant without typestate analysis (w/o TSA), a significant decrease in recall was observed on both benchmarks: a 35.3% decrease on NLBENCH and a 25.5% decrease on EXTERNALBENCH. This finding highlights the substantial contribution of the typestate analysis technique in detecting bugs related to control flow violations.

Conversely, the variant without value propagation (w/o VP) exhibited a more substantial decline in recall on EXTERNALBENCH (36%) compared to NLBENCH (31.2%). This discrepancy can be attributed to the higher prevalence of bugs related to value violations and statements that co-change together in EXTERNALBENCH, which the VP technique is particularly adept at handling.

The ablation study reveals the distinct contributions of each component (i.e., removing either typestate analysis or value propagation reduced recall). These results demonstrate that both techniques positively contribute to NEURALSTATE bug detection performance.

F. Time Complexity Comparison (RQ4)

Experimental Setup. To evaluate the scalability of NEURALSTATE and compare it with NeuraLint, we followed a similar procedure as the one used in the NeuraLint study. We created a set of DL programs with varying numbers of layers: 10, 15, 20, 25, 30, and 35 layers. We then executed these programs using both NEURALSTATE and NeuraLint and measured their respective execution times. The coefficient of determination (R^2) was employed to quantify the goodness of fit, with a value closer to 1 indicating better scalability as the number of layers increases.

Results. NEURALSTATE demonstrates a more efficient execution than NeuraLint, as illustrated in Figure 8. As seen in the figure, NEURALSTATE shows around 12% increase in R^2 value when compared with NeuraLint. The enhanced scalability of NEURALSTATE can be attributed to its approach of using an abstract representation for DL programs, called the NeuralState Sequence. In contrast, NeuraLint relies on the external GROOVE tool for graph checking, as mentioned in their publication [8]. This dependence on an external tool introduces additional computational overhead and communication costs, leading to longer execution times, as the codebase size increases.

VII. DISCUSSION AND THREATS TO VALIDITY

A. Real-World Bug Detection Capabilities

Evaluating a static analysis tool against open-source repositories, such as GitHub, is a valuable method for demonstrating real-world utility. However, this approach presents a unique challenge for the types of bugs that NEURALSTATE detects. Because our tool identifies structural and API usage violations that often cause program crashes or prevent training to start, developers tend to discover and fix these issues locally before committing their code. This makes finding such faults in version histories difficult. Therefore, to rigorously evaluate NEURALSTATE on real-world faults, we turned to established benchmarks curated from Stack Overflow developer questions [8, 19]. On these two independent benchmarks, our approach identified 18 unique bugs that the state-of-the-art tool, NeuraLint, missed entirely. This result confirms that NEURALSTATE is effective in locating real-world bugs that developers struggle with and that existing tools cannot detect.

B. Generalizability to Other Frameworks

Currently, NEURALSTATE supports Keras-based DL programs. While our framework is general, adapting it to other DL frameworks, for instance, to include PyTorch, demands engineering effort. Our tool's initial step involves a precise extraction of layer information from the given DL program. This structural analysis (parsing) component must understand the conventions of each DL framework. For example, Keras provides a convenient `.layers()` API to extract model structure, whereas PyTorch relies on `.named_modules()`. Adapting our extraction mechanism to accurately interpret and translate these distinct framework-specific representations into our tool's abstract model requires looking into the semantic differences of how each DL framework defines and connects computational layers. While our core analytical engine, which employs typestate analysis and value propagation, remains broadly applicable, the component responsible for understanding these varied program structures requires re-engineering. Furthermore, our current version only supports fully connected and convolutional neural networks. Expanding its analysis capabilities to encompass a broader range of network architectures will significantly enhance NEURALSTATE's usefulness for developers and researchers exploring diverse models.

C. Framework Version Compatibility

Our evaluation uses programs developed with Keras 2.x. While newer versions of Keras are available, our findings remain relevant as Keras 3.0 maintains backward compatibility with Keras 2.x [22]. Given that core APIs remain largely consistent, the architectural patterns and bugs identified are fundamental to deep learning [22]. Future investigations could apply these findings to newer or evolving codebases.

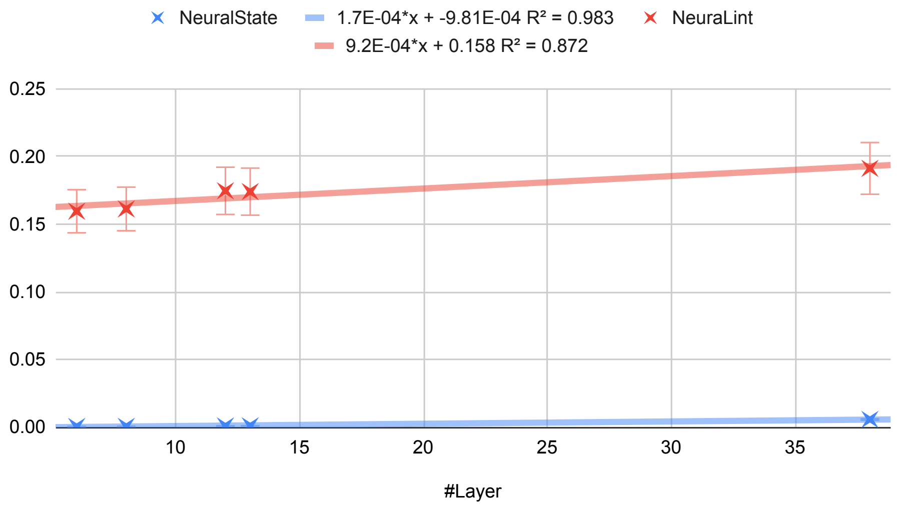


Fig. 8: Time complexity of NEURALSTATE and NeuraLint on detecting DL bugs. The x-axis represents the number of layers and the y-axis represents the time cost (sec)

D. Specification Completeness and Bias

A fundamental challenge in our approach lies in ensuring specification completeness, which has two dimensions. First, regarding the *internal completeness* of our formal model, for every abstract state and every supported DL operation in our grammar (Tables I and II), the Finite State Automaton (FSA) explicitly defines a transition. This ensures that our analysis is deterministic and will always yield a result (either a valid next state or a transition to the explicit error state), avoiding undefined behaviors in the analysis itself.

Second, regarding *external completeness*, our 24 specifications were derived from official Keras documentation and prior empirical studies [8] to cover the most prevalent API protocols. However, we acknowledge that this manual process cannot cover all APIs or newly introduced API constraints. This represents a trade-off: our method provides sound, verifiable detection for a critical set of known bug patterns, at the cost of potentially missing violations of unspecified rules. A promising direction for future work is should focus on semi-automated techniques for mining specifications to improve coverage. Similarly, a limitation of our current FSA is its single error state. We recognize that some specifications, such as the preference for Max-pooling (R6), represent best practice rather than strict, crash-causing violations. In the future, we will enhance our automaton to support multiple fault states (e.g., 'warning' for best practices and 'error' for definite violations), which would provide better context-specific feedback for developers.

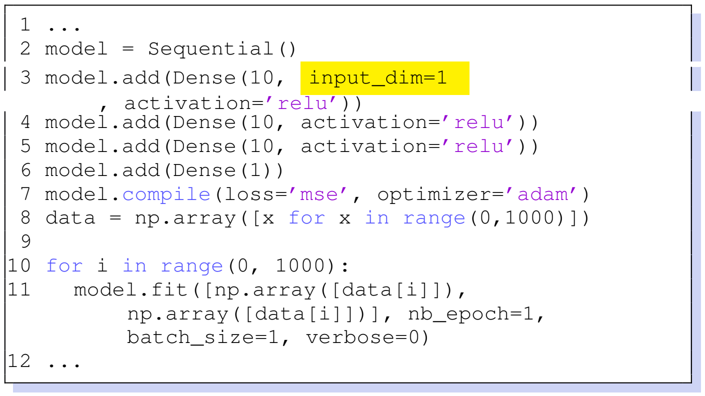
E. Soundness

Our approach prioritizes soundness over completeness. This means that when our method identifies a violation, there is, in fact, a violation. However, there may be instances where our approach fails to detect existing violations, leading to false negatives. Continuous refinement and expansion of the specifications are crucial to improve its ability to detect a wider range of bugs.

F. Internal Threats

Our results indicate a higher false negative (FN) rate for bugs that cause program crashes compared to other bug types. A closer analysis of FN reports reveals that 46% were caused by a mismatch between the input shape expected by the DL model and the actual shape of the training dataset, as observed from Stack Overflow code fixes. This issue arises due to a lack of information about the training dataset, such as the number of classes or the type of task (e.g., classification, regression).

Consider the bug from Stack Overflow shown in Fig. 9. Here, a developer wanted to map 5-element input arrays to 3-element outputs, but they incorrectly configured the first layer to expect one input (`input_dim=1`) and the final layer to produce one output (`Dense(1)`). Our tool, NEURALSTATE, misses this bug entirely because from an architectural view, a sequence of `Dense` layers is valid. The actual error is not in the layer sequence itself, but in the mismatch between the architecture and the `data` array later passed to `model.fit()`.



```
...
model = Sequential()
model.add(Dense(10, input_dim=1
    , activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
data = np.array([x for x in range(0,1000)])

for i in range(0, 1000):
  model.fit([np.array([data[i]]),
      np.array([data[i]])], nb_epoch=1,
      batch_size=1, verbose=0)
...
```

Fig. 9: A Stack Overflow bug that NEURALSTATE cannot detect due to its data-agnostic design.

This case reveals a design trade-off we made in NEURAL-

STATE. It was designed to be light-weight and **data-agnostic**, ensuring that it does not need access to potentially large or sensitive training data. However, the consequence is that the tool is blind to the data-shape mismatches shown in the example. This points to a direction for future work on incorporating data characteristics without trading off efficiency.

VIII. RELATED WORK

A. Detecting Deep Learning Bugs

Various techniques have been proposed to detect and prevent bugs in DL models [4, 6, 8, 9, 23, 24, 25, 26]. Among these methods, [6, 10] focus on detecting performance-related bugs at runtime. Similarly, other approaches, such as [7, 24, 25] monitor the model's training to detect performance bugs on specific symptoms. Recently, machine learning-based approaches like DEfault [27] have extended this by using dynamic runtime features to classify a wide range of both training and model faults.

In contrast, NEURALSTATE employs an analysis that does not require the dataset or model training, thus examining a DL program without running it. Pytee [28] is a static analysis tool designed specifically to detect shape mismatches in PyTorch programs. It converts the Pytorch program into a custom intermediate representation and runs an SMT-solver to detect the bugs. While we recognize the value of Pytee, comparing our tool to it presents some challenges. Pytee specifically supports PyTorch programs, which are not included in our benchmark. Additionally, it addresses only a portion of the bug types that our tool comprehensively covers. Amimla [29] constructs an abstract representation of a DL program and ML pipeline. It then builds a database of DL constraints for symbolic analysis to pinpoint issues like dimension mismatches and incorrect API calls. A direct comparison with Amimla is challenging due to the tool's unavailability. Theoretically, Amimla and NeuralState differ in three aspects: program representation, the specification design, and the type of analysis to identify bugs. First, Amimla separately represents different stages of model-building using graph representation and hash tables without data dependencies. Second, Amimla uses a key-value pair to store valid API usage, whereas NeuralState encodes the specification as a finite state automaton. Lastly, Amimla uses symbolic analysis to identify bugs, and NeuralState uses typestate and value-propagation. The closest openly available work to ours is Neuralint [8]. Neuralint is a static analysis tool proposed by Nikanjam et al. [8]. It utilizes a meta-modeling and graph transformation technique to identify DL bugs. While Neuralint can detect common DL bugs, it reports a high rate of false positives and negatives due to its inability to resolve data dependencies and co-change statements. In comparison, NEURALSTATE runs the analysis on top of a DL representation that accounts for data dependence between statements and uses context-sensitive typestate and value propagation to handle the co-changing statement.

A more recent direction in DL bug detection and repair leverages Large Language Models (LLMs). For example, LLMAPIDet [11] uses an LLM to build a knowledge base of API misuse rules from a large corpus of code commits. Then, for a new code snippet, it uses a few-shot prompting technique to check if any of these natural-language rules apply and, if so, generates a patch. While powerful, this LLM-based approach relies on the probabilistic knowledge of a massive, pre-trained model, which can be non-deterministic and acts as a black box. NEURALSTATE offers a different set of guarantees. Our approach is grounded in formal methods using typestate analysis, thus making our analysis verifiable and deterministic. However, we see these approaches as complementary. The primary manual effort in our work is defining the specification, abstract states, and transition rules for our finite state automaton. The `LLMAPIDet` paper demonstrates a direct solution to infer specification. A promising direction for future work is to leverage their strategy to extract these rules and then translate them into the formal transitions for our typestate automaton. This would create a hybrid approach, combining the broad knowledge-mining of LLMs with verifiable guarantees of our formal analysis.

B. Typestate Analysis

Strom and Yemini [20] first introduced the concept of typestate as a refinement to type systems. CrySL [15] is a notable tool that uses typestate analysis and data-flow analysis to specify usage protocols of cryptographic APIs. While CrySL is shown to be effective at detecting misuse of cryptographic APIs, it cannot be directly applied to languages other than Java as it requires a compiler to convert the rules. Recent studies [16, 17, 30] have also used typestate analysis to detect traditional software vulnerabilities, cloud APIs, and OS bugs. However, none of these approaches consider DL bugs, which exhibit distinct data dependency characteristics.

C. Value analysis

Several approaches [14, 31, 32, 33, 34] use a value-flow analysis technique to detect traditional software bugs. One recent example is the Canary [14] approach, which employs a thread-modular algorithm to capture data and interference dependencies within a value-flow graph, addressing bugs in concurrent programs.

IX. CONCLUSION

In this paper, we present NEURALSTATE, an approach for detecting bugs in deep learning programs that address the limitations of state-of-the-art tools. The key insights behind NEURALSTATE include capturing data dependencies among DL layers, reasoning about complex bug patterns that require simultaneous modifications to multiple statements, and combining typestate analysis with value propagation. Empirical evaluations on two benchmarks containing real-world Keras bugs demonstrate NEURALSTATE's significant improvement over the state-of-the-art tool, NeuraLint. The contributions of this work include the introduction of the NeuralState Sequence (NSS) representation and the development of a technique for handling co-changing statements. Future work aims to explore techniques for automating the construction of automaton specifications and its integration with bug detection tools.

X. Data-Availability

A replication package and benchmark of our tool is available at [35].

XI. Acknowledgments

This work was supported by the National Science Foundation under Grants: 2512857 and 2512858. All opinions are those of the authors and do not reflect the views of sponsors.

References

[1] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Comput. Surv.*, vol. 54, no. 10s, sep 2022. [Online]. Available: https://doi.org/10.1145/3505243

[2] StackOverflow, "Stackoverflow trend for keras," 2024. [Online]. Available: https://trends.stackoverflow.co/?tags=keras

[3] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 510–520. [Online]. Available: https://doi.org/10.1145/3338906.3338955

[4] Pylint, "Pylint 2.16.0b1 documentation," 2003. [Online]. Available: https://pylint.readthedocs.io/en/latest/

[5] A. Gulli and S. Pal, *Deep learning with Keras*. Birmingham, United Kingdom: Packt Publishing Ltd, 2017.

[6] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 561–572.

[7] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, "Deepfd: Automated fault diagnosis and localization for deep learning programs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 573–585. [Online]. Available: https://doi.org/10.1145/3510003.3510099

[8] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, "Automatic fault detection for deep learning programs using graph transformations," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, sep 2021. [Online]. Available: https://doi.org/10.1145/3470006

[9] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: Analysis for machine learning programs," in *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–10. [Online]. Available: https://doi.org/10.1145/3211346.3211349

[10] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE. New York, NY, USA: Association for Computing Machinery, 2021, pp. 251–262.

[11] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and detecting misuses of deep learning apis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[12] eforkin, "Keras Python Multi Image Input shape error," https://stackoverflow.com/questions/44399299, 2017.

[13] K. Bierhoff and J. Aldrich, "Modular typestate checking of aliased objects," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 301–320. [Online]. Available: https://doi.org/10.1145/1297027.1297050

[14] Y. Cai, P. Yao, and C. Zhang, "Canary: Practical static detection of inter-thread value-flow bugs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1126–1140. [Online]. Available: https://doi.org/10.1145/3453483.3454099

[15] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2019.

[16] M. Emmi, L. Hadarean, R. Jhala, L. Pike, N. Rosner, M. Schäf, A. Sengupta, and W. Visser, "Rapid: Checking api usage for the cloud in the cloud," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1416–1426. [Online]. Available: https://doi.org/10.1145/3468264.3473934

[17] T. Li, J.-J. Bai, Y. Sui, and S.-M. Hu, "Path-sensitive and alias-aware typestate analysis for detecting os bugs," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 859–872. [Online]. Available: https://doi.org/10.1145/3503222.3507770

[18] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization to detect co-change fixing locations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 659–671.

[19] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for

Computing Machinery, 2020, p. 1110–1121. [Online]. Available: https://doi.org/10.1145/3377811.3380395

[20] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, Jan 1986.

[21] A. Ghanbari, D.-G. Thomas, M. A. Arshad, and H. Rajan, "Mutation-based fault localization of deep neural networks," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1301–1313.

[22] Keras, "Keras 3.0 documentation," 2023. [Online]. Available: https://www.keras.io/keras_3

[23] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static Analysis of Shape in TensorFlow Programs," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Hirschfeld and T. Pape, Eds., vol. 166. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 15:1–15:29. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13172

[24] E. Schoop, F. Huang, and B. Hartmann, "Umlaut: Debugging deep learning programs using program structure and model behavior," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445538

[25] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 359–371.

[26] R. Manke, M. Wardat, F. Khomh, and H. Rajan, "Leveraging data characteristics for bug localization in deep learning programs," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–29, 2025.

[27] S. Jahan, M. B. Shah, P. Mahbub, and M. M. Rahman, "Improved detection and diagnosis of faults in deep neural networks using hierarchical and explainable classification," *arXiv preprint arXiv:2501.12560*, 2025.

[28] H. Y. Jhoo, S. Kim, W. Song, K. Park, D. Lee, and K. Yi, "A static analyzer for detecting tensor shape errors in deep neural network training code," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 337–338.

[29] M. J. Islam, "Towards understanding the challenges faced by machine learning software developers and enabling automated solutions," Ph.D. dissertation, Iowa State University, 2020.

[30] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 999–1010. [Online]. Available: https://doi.org/10.1145/3377811.3380386

[31] Q. Shi, R. Wu, G. Fan, and C. Zhang, "Conquering the extensional scalability problem for value-flow analysis frameworks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 812–823. [Online]. Available: https://doi.org/10.1145/3377811.3380346

[32] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," *SIGPLAN Not.*, vol. 53, no. 4, p. 693–706, jun 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192418

[33] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 480–491. [Online]. Available: https://doi.org/10.1145/1250734.1250789

[34] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 254–264. [Online]. Available: https://doi.org/10.1145/2338965.2336784

[35] Anonymous, "NeuralState implementation and benchmarks," https://zenodo.org/records/14931679, 2025.