NeuralState: An Approach for Detecting Protocol Violations in a Deep Learning Program

Fraol Batole*, Ruchira Manke *, Robert Dyer † Tien N. Nguyen ‡, and Hridesh Rajan*
* *Department of Computer Science, Iowa State University*, Ames, IA, USA
*{fraol, rmanke, hridesh}@iastate.edu
† *University of Nebraska–Lincoln*, Lincoln, NE, USA, rdyer@iastate.edu
‡ *Computer Science Department, The University of Texas at Dallas*, Dallas, TX, USA, tien.n.nguyen@utdallas.edu

Abstract—**Deep Learning (DL) is widely used in various applications, ranging from healthcare to chatbots. However, violations of usage protocols in these applications can lead to severe consequences, such as inaccurate predictions and system failures. To address this issue, researchers have proposed several techniques for detecting and preventing bugs in deep learning programs. However, existing static analysis tools only focus on undefined variables and shape-related bugs or do not consider a crucial property of DL programs, such as data dependency between layers. To address this challenge, 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 using a real-world benchmark that contains 45 real-world buggy programs shows that NeuralState can precisely detect more bugs than the state-of-the-art tool, NeuraLint. Specifically, it shows a 25% relative improvement in precision and 63% relative improvement in recall when compared with an existing technique.**

I. Introduction

Deep Learning (DL) rapidly transforms various domains, from software engineering to conversational AI [1], [2]. However, the popularity of DL can also bring challenges to developers, notably in the adherence to Application Programming Interface (API) usage protocols. Violations of these protocols can lead to significant issues, such as incorrect predictions or program crashes [3]. For instance, studies indicate that API-related bugs are responsible for up to 16% of performance issues and cause over 66% of program crashes in Keras [3]. These bugs commonly arise from invalid API call sequences or unsuitable input to operations [3].

Popular libraries like TensorFlow, Keras, and PyTorch [4]–[6] provide developers with various DL operations, each having specific API usage protocols. However, these protocols often lack formal specifications, making it challenging for developers to understand and comply with them effectively, especially given their complexity and volume [3].

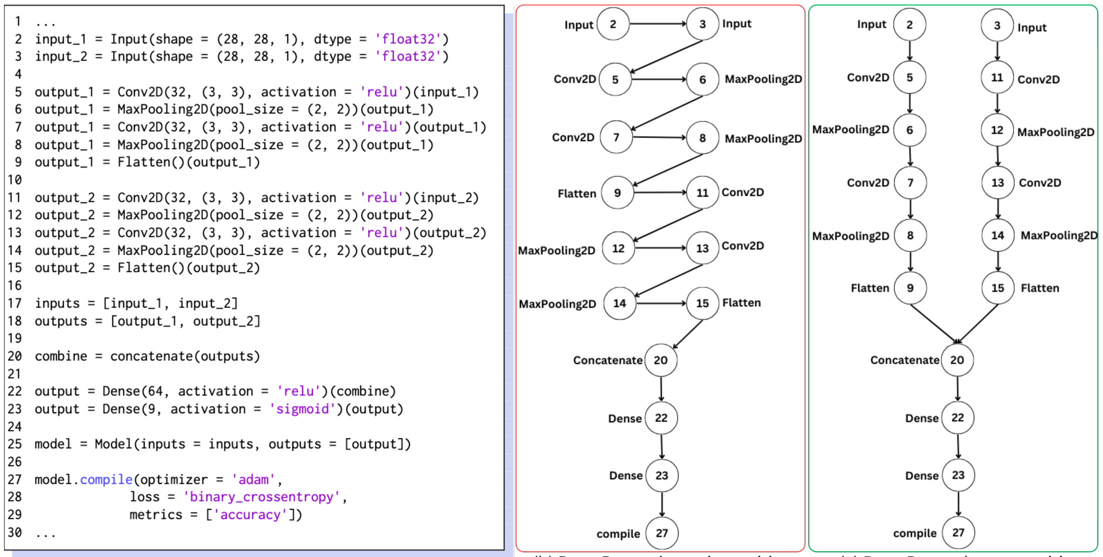
Current state-of-the-art in DL bug detection. Several techniques have been proposed to detect bugs in DL programs [7]–[9]. While some focus on runtime performance issues or tensor shape errors [10], [11], they require access to training data and model training procedures. NeuraLint [9] is a notable static analysis tool for detecting API usage violations. It constructs a control flow graph using intra-procedural analysis and employs a graph-based verification technique to identify DL bugs. However, NeuraLint struggles to handle data dependencies and co-changing statements crucial for precise bug detection [12].

Figure 1 showcases a buggy DL program from Stack Overflow [13] where the final layer uses an incorrect activation function ('sigmoid'). Fixing this bug requires modifying both the activation function to 'softmax' and the loss function to 'categorical_crossentropy'. Such co-dependent changes, where multiple statements require simultaneous modifications, are prevalent in DL programs but often overlooked by existing bug detection tools, like NeuraLint. We refer to these statements as co-changing statements.

NeuralState: our approach for DL bug detection. To address the limitations of existing tools, we propose NeuralState, an approach for detecting performance and program crash bugs in deep learning programs. NeuralState leverages formal specifications of DL operations and combines typestate analysis and value propagation to identify protocol violations. Typestate analysis ensures that the program adheres to the expected sequence of layer operations, while value propagation verifies the validity of inputs to each layer. This combination allows our approach to detect both control flow and data flow errors in DL programs.

NeuralState relies on three key observations.

1) *Capturing Data Dependencies.* DL programs often exhibit data dependencies, where different code segments interrelate through shared data. For example, in the above program, the layers on (lines 5-9) and (lines 11-15) are designed to process different inputs at line 2 and line 3, respectively. Tools, such as NeuraLint that overlook these object-oriented aspects in favor of a procedural perspective can miss the implicit parameters (e.g., '(input_1)' at line 5), thus leading to inaccurate bug reporting. This happens because missing the implicit parameters leads to building a representation akin to Figure 1 (b). This oversight highlights the need for an approach that captures data-dependent statements, thus building a correct representation as shown in Figure 1 (c).
2) *Considering Interdependent Changes.* Fixing protocol violations in DL programs often requires modifying mul-



```
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: Data-dependence-sensitive control flow in NEURALSTATE, instead of data-dependence-insensitive flow in NeuraLint

tiple interdependent statements simultaneously. For example, changing the activation function in the final layer (Figure 1, line 23) necessitates a corresponding change in the loss function (line 28). Effective bug detection must consider these interdependencies to accurately identify and repair protocol violations.

3) *DL Bugs Can Stem from Control Flow or Input Value Violations.* Deep learning programs rely on both the correct sequence of layer operations and the validity of input values. For example, omitting a required layer, such as `Flatten()`, before a `Dense` layer can lead to program crashes. Similarly, providing incompatible input to an activation function, such as using 'sigmoid' instead of 'softmax' for multi-class classification tasks, can result in incorrect predictions or unexpected behavior.

We conducted experiments to evaluate NEURALSTATE on two benchmarks. The results demonstrate a 35.1% improvement in precision and a 19.4% relative gain in recall for NEURALSTATE over the state-of-the-art NeuraLint [9] on the first benchmark. On the second benchmark, NEURALSTATE shows a 15.5% relative improvement over NeuraLint in precision and 107% improvement in recall. An ablation study further outlines the contribution of combining typestate analysis and value propagation as integral to NEURALSTATE's superior performance. Detailed experimental outcomes and comparative analyses are presented in Section VII.

This work makes the following key contributions:

1) We introduce NEURALSTATE, an approach for detecting DL usage protocol violations with a combination of typestate and value propagation techniques.
2) We propose a representation of a DL program that accounts for data dependencies among layers.
3) We propose abstract states representing DL operations and design formal state transition rules based on DL specifications.
4) We evaluated NEURALSTATE against the state-of-the-art static analysis tool for DL.

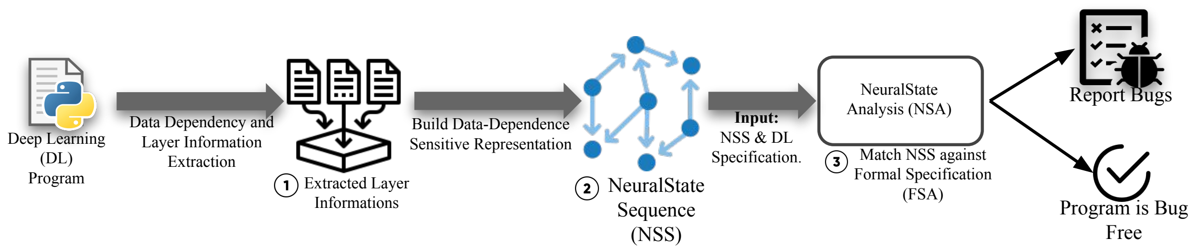
A replication package of our tool is available at [14].

II. PRELIMINARIES

In this section, we discuss the grammar that we propose to capture the common DL operations, describe how we collect specifications, and present our typestate-based approach for formally specifying the behavior of DL programs.

TABLE I: Grammar Representing Supported DL Operations

Symbol	Definition	Description
N	::= L :: N | 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* | *sgd*	Optimizer function of the model
v	::= $x \| x \in \mathbb{Z}^+$	Number of units or neurons
k	::= $(x, y) \| x \in \mathbb{Z}^+,\ y \in \mathbb{Z}^+$	kernel or pooling size
r	::= $x \| x \in \mathbb{R}^+$	Dropout rate



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

A. *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. [9].

B. *Collecting DL Specifications*

We collected DL specifications from prior studies [9], adding an extra rule from the DL library's official documentation (i.e., high dropout rate) [4]. In total, we encode 24 DL specifications using finite-state automaton. Our approach focuses on Fully-Connected Neural Networks (FCNNs) and Convolutional Neural Networks (CNNs) similar to NeuraLint [9]. We have included all specifications as a supplement for further reading.

C. *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 $\delta : S \times L \to S$ 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 deep learning specifications and an analysis of common operations employed in DL models, as documented in prior work [9], 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 [15], [16].

TABLE II: States Representing Phases in DL 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 the features into the desired output format
Compiled	The accepting state, representing the final configuration of the neural network, including the loss function and optimizer

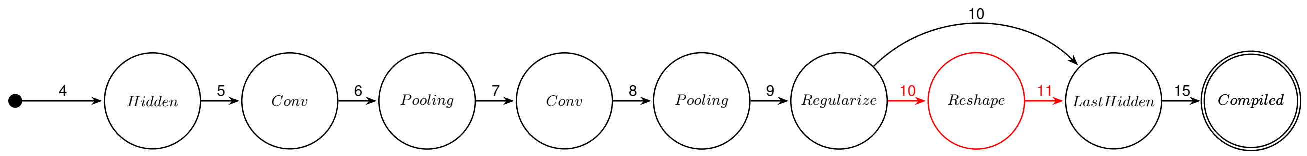
III. APPROACH

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.



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

- **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, while value propagation is employed to identify potential invalid inputs for DL layers. The NSA utilizes the ground truth rules encoded as a finite state automaton (FSA) and the NSS representation of the DL program under analysis.

B. Key Ideas

Based on the observations presented in §I, we introduce the following key ideas:

1) **Data-Dependence-sensitive Control Flow:** we introduce the NSS, which accounts for data dependencies in representing DL programs. The NSS is constructed from the source code based on the grammar of DL operations (see Table I). Then, NSS is matched against the ground truth specification (FSA) to identify DL bugs.
2) **Handling Co-changing Statements:** NEURALSTATE employs context-sensitive analysis to handle co-changing statements, where the correct behavior depends on the context and interactions between multiple statements. This sensitivity allows NEURALSTATE to perform lookups, update values, and track the effects of co-changing statements.
3) **Combining Typestate and Value Propagation Analysis:** NEURALSTATE combines typestate analysis with value propagation to address both control flow and data flow aspects of DL programs. Typestate analysis examines the sequence of layer operations to identify procedural anomalies, while value propagation monitors the validity of input values, ensuring that they conform to the specified constraints.

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 is defined recursively as follows:*

$$NSS = \begin{cases} s\ l :: NSS & \text{if } l \in L \\ s & \text{if } l \in \varnothing \end{cases}$$

where $s \in S$ is an abstract state node, $l \in L$ is a layer operation representing the edge between nodes, and :: denotes the concatenation operator. The base case s represents the terminal state where no further operations are applied.

Intuitively, the NSS represents the sequence of abstract states that a DL program traverses during its execution, with each state transition determined by the corresponding layer operation.

```
 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

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) If the condition in step 2 is met, 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.

1) 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.

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 discusses a high-level overview of state transition rules using an algorithm and presents a precise formulation of state transition rules for DL.

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 ← []
6:      while $s_d \neq \varnothing$ 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 ← Violations ∪ $\{(s_c, s_d)\}$
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, $\Gamma$
```

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).

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

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

$$\frac{\begin{array}{c}\Gamma(Conv) = False \qquad \Gamma(Reshape) = False \\ v \in \mathbb{Z}^+ \quad a_f \in \{relu, tanh\} \quad s_c \in \{Flatten, Dense, Hidden\} \\ \Gamma, Dense \mapsto True = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, } a_f\texttt{)}) = Dense, \Gamma'} \quad \text{(Dense 1)}$$

$$\frac{\begin{array}{c}\Gamma(Reshape) = True \\ v \in \mathbb{Z}^+ \quad a_f \in \{relu, tanh\} \quad s_c \in \{Flatten, Dense, Hidden\} \\ \Gamma, Dense \mapsto True = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, } a_f\texttt{)}) = Dense, \Gamma'} \quad \text{(Dense 2)}$$

$$\frac{\begin{array}{c}l \in \{\texttt{Conv1D(v, k, } a_f\texttt{)}, \texttt{Conv2D(v, k, } a_f\texttt{)}\} \\ v \in \mathbb{Z}^+ \quad k \in \mathbb{Z}^+ \quad a_f = relu \quad s_c \in \{Hidden, Pooling\} \\ \Gamma, Conv \mapsto True, c_v \mapsto v = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, l) = Conv, \Gamma'} \quad \text{(Conv 1)}$$

$$\frac{\begin{array}{c}l \in \{\texttt{Conv1D(v, k, } a_f\texttt{)}, \texttt{Conv2D(v, k, } a_f\texttt{)}\} \\ v \in \mathbb{Z}^+ \quad k \in \mathbb{Z}^+ \quad a_f = relu \quad s_c \in \{Hidden, Pooling\} \\ \Gamma(Conv) = True \qquad \Gamma(c_v) \leq v \qquad \Gamma, c_v \mapsto v = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, l) = Conv, \Gamma'} \quad \text{(Conv 2)}$$

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

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

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

$$\frac{\begin{array}{c}r \leq \alpha \quad s_c \in \{Dense, Conv, Pooling\} \\ \Gamma, Regularize \mapsto True = \Gamma'\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dropout(r)}) = Regularize, \Gamma'} \quad \text{(Regularize)}$$

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

$$\frac{\begin{array}{c}\Gamma(Conv) = False \qquad \Gamma(Reshape) = False \\ 0.8 \geq v \geq 0 \quad a_f \in \{linear, sigmoid, softmax\} \\ s_c \in \{Reshape, Regularize, Normalize\} \\ \Gamma, f \mapsto a_f = \Gamma' \qquad \Gamma', LastHidden \mapsto True = \Gamma''\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, } a_f\texttt{)}) = LastHidden, \Gamma''} \quad \text{(LastHidden 1)}$$

$$\frac{\begin{array}{c}\Gamma(Reshape) = True \\ 0.8 \geq v \geq 0 \qquad a_f \in \{linear, sigmoid, softmax\} \\ s_c \in \{Reshape, Regularize, Normalize\} \\ \Gamma, f \mapsto a_f = \Gamma' \qquad \Gamma', LastHidden \mapsto True = \Gamma''\end{array}}{\Gamma \vdash \delta(s_c, \texttt{Dense(v, } a_f\texttt{)}) = LastHidden, \Gamma''} \quad \text{(LastHidden 2)}$$

$$\frac{\begin{array}{c}\Gamma(f) \in \{linear, sigmoid\} \qquad \Gamma \vdash l_f = binary\_crossentropy \\ o \in \{adam, sgd\} \qquad \Gamma, Compiled \mapsto True = \Gamma'\end{array}}{\Gamma \vdash \delta(LastHidden, \texttt{Compile(}l_f\texttt{, o)}) = Compiled, \Gamma'} \quad \text{(Compiled 1)}$$

$$\frac{\begin{array}{c}\Gamma(f) = softmax \qquad \Gamma \vdash l_f = categorical\_crossentropy \\ o \in \{adam, sgd\} \qquad \Gamma, Compiled \mapsto True = \Gamma'\end{array}}{\Gamma \vdash \delta(LastHidden, \texttt{Compile(}l_f\texttt{, o)}) = Compiled, \Gamma'} \quad \text{(Compiled 2)}$$

Fig. 5: State Transition Rules for DL Specifications

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.

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 `Dense` rule enforces constraints on the activation function and the number of units. To illustrate how the rules are read, consider the second `Dense` rule: "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. NEURALSTATE handles these co-changing statements through the `LastHidden` and `Compiled` rules. The `LastHidden` 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 `Compiled` rule, the analysis consults the recorded activation function f to verify that it matches the appropriate 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.

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 research 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

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 [9] and contains 26 real-world Keras buggy programs collected from Stack Overflow (SO). The second benchmark, **HumbatovaBench**, is taken from a prior study by Humbatova et al. [17] and consists of 19 real-world buggy programs with complete Keras code collected from SO and GitHub posts. In total, the benchmarks contain 22 bugs with a program crash symptom, 21 with bad performance, and 2 with incorrect functionality symptoms.

Evaluation Metric. We adopted the same evaluation metrics as in NeuraLint: Precision = $\frac{\text{TP}}{\text{TP + FP}}$ and Recall = $\frac{\text{TP}}{\text{TP + FN}}$, where TP: True Positive, FP: False Positive, and FN: False Negative. For evaluating the time complexity, we utilize the curve fitting method to obtain a model representing the data, as used in related works [18]. 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.

Baselines. We compared our approach against the state-of-the-art DL bug detection tool, NeuraLint [9]. NeuraLint builds a graph representation of DL programs and runs a graph-based verification tool to capture the bugs.

Implementation. NEURALSTATE is implemented using Python. We use the NetworkX library to build the NeuralState Sequence and the Matplot library to visualize it. We used the

TABLE III: Performance of NEURALSTATE on NLBench (RQ1-A), where PC=Program Crash, BP=Bad Performance, IF=Incorrect Functionality. Differences are emphasized with green and orange colors.

Prog	SO ID	Symptom	SO Fix + Specification Violations	NeuralState			NeuraLint		
				TP	FP	FN	TP	FP	FN
1	44399299	PC	Change the shape of the input layer, Use softmax instead of sigmoid	1	0	1	0	1	2
2	43464835	PC	Change the shape of the input layer	0	0	1	0	0	1
3	42913869	PC	Change the number of units for the output layer, Change the shape of the input layer	0	0	2	0	0	2
4	48518434	PC	Reduce the spatial size of both Conv. filtering and pooling widows	0	0	1	0	0	1
5	40857445	PC	Add a flatten layer, the last layer missing	2	0	0	0	0	2
6	50555434	BP	Use softmax activation instead of sigmoid and categorical_crossentropy loss instead MAE	2	0	0	1	0	1
7	46177505	PC	Change spatial size of Conv. filtering and pooling widows	0	0	3	0	0	3
8	50426349	PC	Change the shape of the input layer	2	0	1	2	0	1
9	38584268	PC	Add a flatten layer	2	0	0	2	1	0
10	45120429	PC	Change the number of units for the output layer, Add a flatten layer	3	0	0	3	0	0
11	45378493	IF	Use a sigmoid for last layer activation	4	0	0	4	1	0
12	45711636	PC	Use channels_last format for input data	0	0	1	0	0	1
13	34311586	BP	Remove the last layer activation	2	0	0	2	1	0
14	50079585_1	BP	Use softmax activation instead of sigmoid and categorical_crossentropy loss instead of binary_crossentropy	1	0	0	1	0	0
15	50079585_2	IF	Change the number of units for the output layer	1	0	0	1	0	0
16	51749207	BP	Use of softmax activation instead of sigmoid	1	0	0	1	2	0
17	53119432	PC	Add a flatten layer	1	0	0	1	1	0
18	55731589	PC	Use of “same” instead of “valid” for layer padding type	0	0	1	0	0	1
19	58844149	BP	Use of softmax activation instead of sigmoid	2	0	0	1	1	1
20	61030068	PC	Add a flatten layer	1	0	0	1	0	0
21	33969059	BP	Change the number of units for the output layer	1	0	0	1	0	0
22	44184091	PC	Fix the limit size for input sequence data	1	0	1	1	0	1
23	44322611	BP	Prune the DNN, use RMSprop instead of SGD	3	0	0	2	0	1
24	49117607	PC	Reduce the spatial size of both Conv. filtering and pooling widows	1	0	0	1	0	0
25	55776436	BP	Try Data augmentation, Regularization, filtering spatial size reduction, and DNN Depth Increase	4	0	0	4	0	0
26	60566498	BP	Try Data augmentation and Hyperparameters Tuning	2	0	1	2	0	1
			TOTAL	**37**	**0**	**13**	**31**	**8**	**19**

NEURALSTATE demonstrates higher effectiveness when detecting bugs that have data dependencies (i.e., rows 1, 5), as well as the bugs that have co-change statements (i.e., rows 6, 11, 13, 16, 19)

Keras library to extract the layer names, as the collected benchmarks are based on it. All the experiments were conducted on a 3.6 GHz 8-Core 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.

VII. EMPIRICAL RESULTS

This section presents the results of our experiments.

A. Effectiveness Evaluation (RQ1)

1) Comparison on NLBench (RQ1-A):

Experimental Setup. We evaluate NEURALSTATE on NLBench.

Detailed Analysis. The detailed results of NEURALSTATE's performance on NLBench are presented in Table III. NEURALSTATE correctly identifies 37 TP out of 50 bugs, while NeuraLint identifies 31 TP. Moreover, NEURALSTATE reports no FP compared to NeuraLint's 8 FP. This substantial reduction in FP demonstrates NEURALSTATE's ability to minimize false alarms, showing a prioritization of soundness over completeness.

The experimental results on NLBench demonstrate NEURALSTATE's effectiveness in detecting a wide range of DL bugs, including those related to activation functions, missing layers, and performance issues. The tool's data-dependence-sensitive analysis and value propagation techniques enable it to capture data dependencies and identify bugs that are often missed by existing tools.

Overview of Results. Table V 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 HumbatovaBench (RQ1-B):

Experimental Setup. To evaluate the generalization capability of our approach (NEURALSTATE) on a broader range of real-world bugs, we conducted an empirical evaluation on an unseen dataset curated by Humbatova et al. [17].

Detailed Analysis. Table IV presents the detailed results of NEURALSTATE's performance in detecting usage protocol

TABLE IV: Performance of NeuralState on HumbatovaBench (RQ1-B), where PC=Program Crash, BP=Bad Performance

Prog	SO ID	Symptom	SO Fix + Specification Violations	NeuralState			NeuraLint		
				TP	FP	FN	TP	FP	FN
1	34716454	PC	Change batch norm layer position, Change softmax function, input shape missing	2	0	1	0	0	3
2	37213388	BP	Change optimizer, DNN Depth Increase	1	0	1	1	0	1
3	41823068	BP	Use softmax, Add regularization	2	0	0	1	0	1
4	45793856	BP	Use sigmoid and binary_crossentropy loss, pooling missing, Change the spatial size of Conv. filtering	4	0	0	0	0	4
5	47272383	BP	Try to increase your Network Depth, Removing the Batch Normalisation, remove the Dropout layer	2	0	1	2	0	1
6	36392966	PC	Change the shape of the input layer	0	0	1	0	0	1
7	45499757	PC	Change the shape of the input layer, DNN Depth Increase	1	0	1	1	0	1
8	43314810	PC	Change the shape of the input layer, Use sigmoid and binary_crossentropy, Flatten missing, Pooling missing	4	0	1	3	0	2
9	47324571	PC	Change the shape of the input layer	0	0	1	0	0	1
10	40369951	BP	Use a lower learning rate	0	0	1	0	0	1
11	43944981	PC	Change the shape of the input layer, Change the spatial size of Conv. filtering	1	0	1	1	0	1
12	48325272	BP	Use softmax and categorical_crossentropy loss instead of MSE	1	0	1	0	0	2
13	46292203	PC	Change the shape of the input layer	1	0	0	0	0	1
14	46642627	BP	Change kernel_initializer	0	0	1	0	0	1
15	48385830	BP	Use softmax and categorical_crossentropy loss instead of MSE	2	0	1	1	0	2
16	37624102	BP	Use a lower learning rate	0	0	1	0	0	1
17	41999686	BP	Use softmax and categorical_crossentropy loss instead of MSE	2	0	0	1	1	1
18	44493395	BP	Change loss to categorical_crossentropy, Regularization	2	0	0	2	0	0
19	50914860	BP	Use softmax and categorical_crossentropy loss	2	0	0	0	1	2
			TOTAL	**27**	**0**	**13**	**13**	**2**	**27**

NeuralState demonstrates more effectiveness when detecting bugs that have high data dependencies (i.e., row 1), as well as bugs that have co-change statements (i.e., rows 3, 4, 8, 12, 15, 17, 19)

violations on HumbatovaBench.

A closer examination of the results reveals that NeuralState effectively detects bugs with data dependencies and those involving co-changing statements. For instance, in row 1, NeuralState correctly identifies two out of three bugs related to incorrect layer positions and invalid activation functions, while NeuraLint fails to detect any of these bugs. Similarly, in rows 3, 4, 8, 12, 15, 17, and 19, NeuralState outperforms NeuraLint in detecting bugs that require simultaneous change to multiple statements, such as changing the activation function and the corresponding loss function.

These findings underscore the importance of NeuralState's data-dependence-sensitive analysis and its ability to handle co-changing statements.

Overview Results. As illustrated in Table V, NeuralState achieved 100% precision and 67.5% recall on HumbatovaBench, 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: Precision and Recall Results

Approach	NLBench		HumbatovaBench	
	Precision	**Recall**	**Precision**	**Recall**
NeuraLint [9]	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%)

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

To illustrate the limitations, we examine concrete examples:

1) **Case study with Program 16 in Table III (Program Crash):** Figure 6 shows a buggy program from the NeuraLint's [9] benchmark. It uses the sequential API to define 6 layers of convolutional and dense operations. The bug cause

```
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. 6: NeuraLint false positive report (Program 16, Table III)

is the use of incorrect activation and loss functions. 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.

TABLE VI: True Positive Overlapping Analysis

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

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' → 'softmax') and on line 28 ('binary_crossentropy' → '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 deep learning code.

C. 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 VII: Impact of Analysis Techniques on NeuralState

Approach	NLBench Precision	NLBench Recall	HumbatovaBench Precision	HumbatovaBench 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 VII presents the results of the ablation study, comparing the performance of NEURALSTATE with its two variants on both benchmarks, NLBench and HumbatovaBench. 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 HumbatovaBench. 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 HumbatovaBench (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 HumbatovaBench, which the value propagation technique is particularly adept at handling.

Overall, the ablation study validates our observation that the combination of value propagation and typestate analysis techniques effectively detect a diverse range of bugs with different characteristics, contributing to NEURALSTATE's better bug detection performance compared to existing tools.

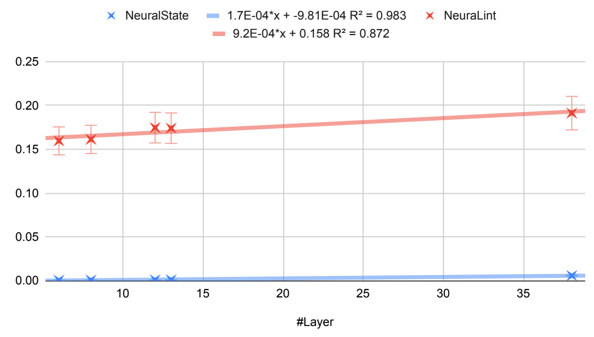


Fig. 7: Time complexity of NEURALSTATE and NeuraLint on detecting DL bugs. The x-axis represents the number of layers and the y-axis represents the time cost (sec)

D. 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 deep learning 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 7. 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 deep learning programs, called the NeuralState Sequence. In contrast, NeuraLint relies on the external GROOVE tool for graph checking, as mentioned in their publication [9]. This dependence on an external tool introduces additional computational overhead and communication costs, leading to longer execution times, as the codebase size increases.

VIII. Limitations and Threats to Validity

Generalizability. Currently, our approach supports Keras-based DL programs. Adapting it to other frameworks like PyTorch may require additional effort to account for API differences. Additionally, NeuralState only supports FCNN and CNN. Extending it to support other network architectures would broaden its applicability.

Specification Bias. Our approach utilizes DL specifications and benchmarks from prior work [9], which may introduce bias and limit the detection of certain bugs. To evaluate the bias of our approach, we include an empirical evaluation on unseen benchmark by Humbatova et al. [17]. However, in the future, we plan to investigate automated or semi-automated approaches to infer specifications from diverse codebases, reducing potential bias and improving coverage.

Soundness or Completeness. 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.

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). To address this limitation, we plan to investigate the impact of incorporating training dataset information and explore methods to integrate it into our analysis.

IX. Related Work

Here, we discuss the related studies on detecting deep learning usage protocol violations, typestate analysis, and value analysis.

Detecting Deep Learning Bugs. Various techniques have been proposed to detect and prevent bugs in DL models [7], [9], [10], [19]–[22]. Among these methods, [7], [11] focus on detecting performance-related bugs at runtime. Similarly, other approaches, such as [8], [20], [21] monitor the model's training to detect performance bugs on specific symptoms. In contrast, NeuralState employs an analysis that does not require the dataset or model training. Amimla [23] 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 [9]. Neuralint is a static analysis tool proposed by Nikanjam et al. [9]. 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.

Typestate Analysis. Strom and Yemini [16] first introduced the concept of typestate as a refinement to type systems. CrySL [24] 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 [25]–[27] 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.

Value analysis. Several approaches [18], [28]–[31] use a value-flow analysis technique to detect traditional software bugs. One recent example is the Canary [18] approach, which employs a thread-modular algorithm to capture data and interference dependencies within a value-flow graph, addressing bugs in concurrent programs.

X. 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.

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] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

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

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. New York, NY, USA: Curran Associates, Inc., 2019.

[7] 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.

[8] 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

[9] 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

[10] 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

[11] 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.

[12] 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.

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

[14] Anonymous, "NeuralState implementation and benchmarks," https://zenodo.org/records/10858991, 2024.

[15] 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

[16] 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.

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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.

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

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

[24] 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.

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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