Refty: Refinement Types for Valid Deep Learning Models

Yanjie Gao
Microsoft Research
China
yanjga@microsoft.com

Zhengxian Li*
Microsoft Research
China
v-zhli20@microsoft.com

Haoxiang Lin†
Microsoft Research
China
haoxlin@microsoft.com

Hongyu Zhang
The University of Newcastle
Australia
hongyu.zhang@newcastle.edu.au

Ming Wu
Shanghai Tree-Graph Blockchain
Research Institute, China
ming.wu@confluxnetwork.org

Mao Yang
Microsoft Research
China
maoyang@microsoft.com

ABSTRACT

Deep learning has been increasingly adopted in many application areas. To construct valid deep learning models, developers must conform to certain computational constraints by carefully selecting appropriate neural architectures and hyperparameter values. For example, the kernel size hyperparameter of the 2D convolution operator cannot be overlarge to ensure that the height and width of the output tensor remain positive. Because model construction is largely manual and lacks necessary tooling support, it is possible to violate those constraints and raise type errors of deep learning models, causing either runtime exceptions or wrong output results. In this paper, we propose Refty, a refinement type-based tool for statically checking the validity of deep learning models ahead of job execution. Refty refines each type of deep learning operator with framework-independent logical formulae that describe the computational constraints on both tensors and hyperparameters. Given the neural architecture and hyperparameter domains of a model, Refty visits every operator, generates a set of constraints that the model should satisfy, and utilizes an SMT solver for solving the constraints. We have evaluated Refty on both individual operators and representative real-world models with various hyperparameter values under PyTorch and TensorFlow. We also compare it with an existing shape-checking tool. The experimental results show that Refty finds all the type errors and achieves 100% Precision and Recall, demonstrating its effectiveness.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

deep learning, validity checking, type error, refinement type

*Work performed during the internship at Microsoft Research.
†Corresponding author.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510077

ACM Reference Format:
Yanjie Gao, Zhengxian Li, Haoxiang Lin, Hongyu Zhang, Ming Wu, and Mao Yang. 2022. Refty: Refinement Types for Valid Deep Learning Models. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510077

1 INTRODUCTION

In recent years, deep learning (DL) has been successfully applied to many application areas such as image recognition, gaming, and natural language processing. To design layered data representations called *deep learning models* (aka deep neural networks) [9, 24], developers employ tensor-oriented mathematical operations provided by deep learning frameworks such as PyTorch [56] and TensorFlow [1]. These operations are known as *operators* that manipulate one or more *tensors* (*i.e.*, multi-dimensional arrays), including, for example, matrix multiplication and 2D convolution. Developers use additional arguments called *hyperparameters* (*e.g.*, the batch size and the kernel size) to control the model learning process.

Like functions in conventional programming languages, operators also enforce *computational constraints* on both tensors and hyperparameters to construct *valid* DL models. Let us take `Conv2d` [61, 74], the 2D convolution operator, as an example for illustration (a complete description is presented in Section 3.2). The input tensor of `Conv2d` should have exactly four dimensions: batch (N), channel (C), height (H), and width (W). Depending on the position of the channel dimension, `NCHW` (channels first) and `NHWC` (channels last) [44, 52] are two widely used tensor formats. PyTorch accepts `NCHW` only [61]; TensorFlow supports both, but developers need to explicitly specify the actual format. The kernel size, a hyperparameter of `Conv2d` that "specifies the height and width of the 2D convolution window" [74], is an array of two positive integers. Besides, the values cannot be overlarge to ensure that the height and width of the output tensor remain positive too. Because model construction is largely manual and lacks necessary tooling support, it is possible to violate the above constraints and raise *type errors* of DL models just like those in conventional programs, causing either a training/inference job to crash (*e.g.*, an overlarge kernel size) or a model to produce totally wrong output results (*e.g.*, `NHWC` training data being fed to a PyTorch model by mistake).

Recent empirical studies [30, 85, 87] indicate that type errors of DL models are not uncommon. For example, Zhang *et al.* [87] discovered that 24 out of the 175 (about 13.71%) TensorFlow program bugs collected from Stack Overflow and GitHub were caused by

incompatible *tensor shapes* (*i.e.*, the lengths of all dimensions). It is challenging to detect and eliminate these errors before job execution (at compile-time) because the hybrid programming paradigm of DL frameworks hides the internal computation from high-level programs written by developers. The type errors of DL models not only waste significant shared resources (including CPU, GPU, network I/O, and storage) but also severely slow down development productivity. Their ill effects are even worse in the widely adopted practice of automated machine learning (AutoML), where an experiment launches many trial jobs simultaneously. That is to say, if one trial job encounters a type error, other tens or hundreds of the jobs with similar neural architecture and *hyperparameter vector* (*i.e.*, a value tuple of all hyperparameters) could also experience the same error and fail.

A simple workaround is to run DL jobs for a while and check whether any runtime exception is thrown. However, such a dynamic method is both resource- and time-consuming; it is especially unaffordable in the AutoML scenario because a large number of possible neural architectures and hyperparameter vectors exist. Furthermore, invalid DL models that do not lead to job crashes but produce wrong output results cannot be caught. Type-based techniques [2, 6, 17, 27, 32, 33, 37, 42, 45, 50, 67, 81] have been promising to detect type errors. However, these previous works target programs written in conventional programming languages such as Haskell, C, C++, or Java; therefore, we cannot apply them directly to DL models because of the wide differences in representation structure. Recently, a few tools [15, 40, 78] such as Pythia [40] are able to find some shape incompatibility errors in certain DL programs. Nevertheless, their approaches are TensorFlow-specific and cannot capture non-shape issues.

In this paper, we propose Refty, a *refinement type*-based tool for statically checking the validity of deep learning models. Refinement types [21] are types endowed with logical formulae that constrain values; for example, $\texttt{int}\{v : 0 < v\}$ stands for positive integers. We observe that the algorithmic execution of a DL model can be represented as iterative forward and backward propagation on the model's *computation graph* [24] whose nodes denote operators. Our key insight is to refine each type of DL operator with logical formulae that describe the computational constraints on both tensors and hyperparameters, including those on how the output tensors are produced. These logical formulae are framework-independent, being formulated from the mathematical definitions of operators. To check whether a PyTorch or TensorFlow model is valid, developers provide its neural architecture (which is described in the Protocol Buffers [25] language or given from a serialized model file) and hyperparameter domains. Refty then traverses the computation graph in strict accordance with the operator execution ordering and generates a set of constraints [28] from the above logical formulae. Therefore, the problem of checking model validity is reduced to a constraint satisfaction problem (CSP) [64]. Refty utilizes a satisfiability modulo theories (SMT) [12] solver (*e.g.*, Microsoft Z3 [11]) to obtain the unsatisfiable hyperparameter vectors that result in potential type errors. To accelerate constraint solving, we apply some special optimization techniques. Refty is extensible to incorporate new operators and custom tensor shapes/element types/formats; it can also be adapted to other DL frameworks.

```python
import torch.nn as nn
class CNNModel(nn.Module):
  def __init__(self):
    super(CNNModel, self).__init__()
    self.conv = nn.Conv2d(in_channels = 1, out_channels = 16, kernel_size = 3)
    self.pool = nn.AvgPool2d(kernel_size = 2, stride = 2)
    self.fc = nn.Linear(in_features = 2704, out_features = 10)
    self.softmax = nn.Softmax(dim = 1)

  def forward(self, x):
    x = self.conv(x)
    x = self.pool(x)
    x = x.reshape(x.size(0), -1)
    x = self.fc(x)
    x = self.softmax(x)
    return x
```

Figure 1: A sample PyTorch model constructed with the `Conv2d`, `AvgPool2d`, `Reshape`, `Linear`, and `Softmax` operators.

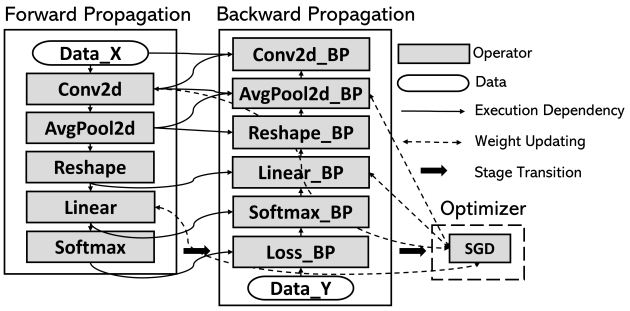

Figure 2: Computation graph for training the above model.

We have implemented Refty and evaluated it on both individual DL operators (`Conv2d`, `MaxPool2d`, `Linear`, `Add`, and `Concat`) and representative real-world models (AlexNet [39], VGG-16 [69], Inception-V3 [71], LSTM [29]-based Seq2Seq [70], and GRU [8]-based Seq2Seq) under the PyTorch and TensorFlow frameworks. We also compare Refty with Pythia, a static shape-checking tool for TensorFlow Python programs. The experimental results show that Refty finds all the type errors, achieves 100% Precision and Recall [51], and outperforms Pythia, demonstrating its effectiveness.

In summary, this paper makes the following contributions:

(1) We propose a novel refinement type-based approach for checking the validity of DL models ahead of job execution.
(2) We implement a tool named Refty that generates a set of constraints for DL models and utilizes an SMT solver to detect potential type errors.
(3) We demonstrate the practical effectiveness of Refty with a rich set of experiments.

2 BACKGROUND

2.1 Deep Learning Models

A deep learning (DL) model is a layered data representation learned from massive training data [9, 24]. The model is formalized by frameworks like PyTorch [56] and TensorFlow [1] as a *computation graph* (*i.e.*, a tensor-oriented, directed acyclic graph) [24]. Each graph node denotes a mathematical operation called an *operator* that manipulates a list of tensors. The node may contain numerical learnable parameters (*i.e.*, weights and biases), which are iteratively updated during the model learning process. A directed edge from node A to another node B delivers one output tensor of A to B as

Table 1: Common type errors of deep learning models.

Dimension	Category	Description
Hyperparameter	Illegal Value	The value of a hyperparameter is outside the domain. For example, typical hyperparameters such as the batch size and stride are positive by definition, but zero or negative values are passed.
	Improper Value	The value of a hyperparameter does not meet the computational constraints. For example, the stride mistakenly exceeds the kernel size of `Conv2d`, causing some input data to be skipped.
Tensor	Unsupported Format	The format of a tensor is not supported. For example, an `NHWC` tensor is fed to the PyTorch `Conv2d`.
	Illegal Shape	A certain dimensional length of a tensor is zero or negative. For example, if the kernel size of `Conv2d` is too large, the output height or width may be reduced to a non-positive value.
	Incompatible Shape	The shape of a tensor is unacceptable, or multiple tensors do not have matched shapes. For example, the input tensor of `Conv2d` does not have exactly four dimensions.
	Incompatible Element Type	A tensor's element type is unacceptable, or multiple tensors do not have matched element types.

input and specifies their execution dependency. In this paper, the terms "node" and "operator" are used interchangeably.

A *tensor* is a multi-dimensional array of numerical values of the same element type. Its *order* (aka rank) is the number of dimensions [55]. Let $\mathbb{R}_f$ denote the set of all 32-bit floating-point numbers. Suppose that $\mathcal{X}$ is an n-order, 32-bit floating-point tensor, and I_i is the length (a positive integer) of its i-th dimension where $1 \leq i \leq n$. Then, $\mathcal{X} \in \mathbb{R}_f^{I_1 \times I_2 \times \cdots \times I_n}$. The array $[I_1, I_2, \cdots, I_n]$ is called the *shape* of $\mathcal{X}$. For example, "$[16, 1, 28, 28]$" is the shape of a 4-order, `NCHW` tensor representing sixteen images of handwritten digits [14]. Because each dimension of $\mathcal{X}$ may have application-specific semantics (*e.g.*, representing the batch size or the image width), dimensional orderings derive different *tensor formats*. For instance, `NCHW` (channels first) and `NHWC` (channels last) [44, 52] are two widely used formats for a batch of 2D images.

Figure 1 shows a simple PyTorch training program, which sets up a sequential model with the framework built-in `Conv2d` (2D convolution with a 3×3 kernel size), `AvgPool2d` (2D average pooling with 2×2 kernel size and stride), `Reshape` (flattening the input into one dimension without affecting the batch size), `Linear` (fully connected layer with ten output features), and `Softmax` (normalizing "the probability distribution over k different classes" [24]) operators (lines 6–9, 14). The variables such as `kernel_size`, `stride`, and `out_features` are *hyperparameters* since they participate in the computation of operators to control the learning process. To obtain the optimal model learning performance (*e.g.*, predictive accuracy), developers largely adopt a trial-and-error strategy by running multiple jobs, each with a different value tuple of all hyperparameters. Figure 2 demonstrates the corresponding computation graph for training the model. The operators on the left are specified by developers in the training program. The auxiliary operators in the middle are automatically crafted by frameworks for computing gradients under backward propagation. An optimizer at the bottom right is responsible for weight update and loss minimization, marking the end of one training iteration.

Although the model is simple enough, developers may make the same mistakes mentioned in Section 1 at `Conv2d` and `AvgPool2d` (lines 5–6). Furthermore, developers may pass a wrong number of input features to the `Linear` operator (via the `in_features` parameter in line 7) and raise an incompatible-tensor-shape error, because such a number has to be manually calculated from the output tensor shapes of the predecessor `AvgPool2d` and `Conv2d`.

2.2 Common Type Errors of DL Models

Table 1 lists six categories of common type errors, which are summarized from related empirical studies [30, 85–87] and our experience. We further group these categories into two major dimensions: Hyperparameter and Tensor. Errors in the former dimension are directly caused by hyperparameter values. *Illegal Value* means that the value of a hyperparameter is outside the domain. Typical hyperparameters are positive integers by definition; however, since they are declared as signed integers in Python, developers may pass zero or negative values by mistake. For example, a user encountered a crash when setting the stride to zero [59]. *Improper Value* means that the value of a hyperparameter violates the intrinsic computational constraints, although it is within the domain. For instance, the stride should be less than or equal to the kernel size; otherwise, some input data will be skipped [72]. Another example is that an out-of-bounds exception is thrown when the axis (specified by `dim`) of the PyTorch `Gather` operator exceeds the input tensor's order.

Tensor-related errors result from inappropriate formats, shapes, or element types of tensors. *Unsupported Format* means that the format of a tensor is not supported by the operator, which is demonstrated by the previous `NHWC` vs. `NCHW` example. Because the output shape is computed by the input shape(s) and hyperparameters, it is possible that a certain dimensional length of an output tensor becomes less than or equal to zero and raises an *Illegal Shape* error. For instance, if the kernel size of `Conv2d` is too large, the output height or width may be reduced to a non-positive value [54]. *Incompatible Shape* means that the shape of a tensor is unacceptable or multiple tensors do not have matched shapes. This category is also referred to as *Unaligned Tensor*, *Mismatched Tensor*, or *Shape Inconsistency* in the related studies. ShapeFlow [78] shows an example that the developer mistakenly interchanged the images and their labels. Another example is that the two input tensors of the addition operator do not have exactly the same shape. Similarly, errors in the *Incompatible Element Type* category are caused by unacceptable or mismatched element types.

2.3 Refinement Types

In programming languages, the type system [57] is an important and useful formal method to enforce the expected behaviors of programs. A type is an attribute of data that permits developers to specify correct values for data operations. Therefore, a large portion of unintended software faults (aka type errors) occurring at run-time can be detected by compilers before execution.

$$\texttt{MatMul} :: x_1 : ts\{x_1 : p_{x_1}\} \rightarrow x_2 : ts\{x_2 : p_{x_2}\} \rightarrow ts\{y : p_y\}$$
$$p_{x_1} \doteq (x_1.order = 2) \wedge (0 < x_1.shape[0]) \wedge (0 < x_1.shape[1])$$
$$p_{x_2} \doteq (x_2.et = x_1.et) \wedge (x_2.order = 2) \wedge (x_2.shape[0] = x_1.shape[1]) \wedge (0 < x_2.shape[0]) \wedge (0 < x_2.shape[1])$$
$$p_y \doteq (y.et = x_1.et) \wedge (y.order = 2) \wedge (y.shape[0] = x_1.shape[0]) \wedge (y.shape[1] = x_2.shape[0])$$

Figure 3: Refinement type of the `MatMul` operator.

In 1991, Freeman and Pfenning [21] first introduced the concept of refinement types to Standard Meta Language (SML) [49], a general-purpose functional programming language. A refinement type lets developers endow logical formulae (from a decidable logic) to limit the value set of the original type. For instance, the following `nat` and `pos` refinement types specify non-negative integers and positive integers, respectively:

$$\texttt{nat} = \texttt{int}\{v : 0 \leq v\}, \qquad \texttt{pos} = \texttt{int}\{v : 0 < v\}.$$

For another example, a finite set of positive integers $\{n_i \mid n_i \in \mathbb{N} \wedge 0 < n_i \text{ for } i \in [0, k]\}$ can be specified by the refinement type $\texttt{pos}\{v : v = n_1 \vee v = n_2 \vee \cdots \vee v = n_k\}$.

With refinement types, developers can write more precise *contracts* [47] for programs to improve code testing and verification. In the following, we take the `MatMul` (matrix multiplication) operator as an example and illustrate its refinement type in Figure 3. Let ts be the basic tensor type. For a tensor variable x, we use $x.et$, $x.order$, and $x.shape$ to denote the element type (*e.g.*, `float` or `double`), order, and shape of x, respectively. $x.shape[i]$ refers to the length of the i-th dimension of x. We ascribe `MatMul` a function type, specifying that it accepts two input tensor parameters (denoted by x_1 and x_2) and produces one output tensor parameter (denoted by y). The logical formula p_{x_1} states that x_1 is a legal matrix; that is to say, the order of x_1 is 2, and the length of either dimension is positive. p_{x_2} declares that x_2 is a legal matrix with the same element type as x_1, and the number of rows of x_2 is equal to the number of columns of x_1. Finally, p_y claims that the output tensor y is also a matrix with the same element type as the two inputs, and y has the number of rows of x_1 and the number of columns of x_2. To save space, we do not explicitly assert the legality of y, which has been implied by the existing formulae.

2.4 Satisfiability Modulo Theories (SMT)

In the fields of mathematical logic and computer science, satisfiability modulo theories (SMT) [12] refers to deciding the satisfiability of mathematical formulae. Compared with Boolean satisfiability (SAT), whose formulae are built up from only Boolean variables and logical connectives (*e.g.*, negation and conjunction), SMT significantly generalizes the expressiveness by allowing more complex first-order formulae with equality, quantifiers ($\forall$ and $\exists$), and symbols of constants (*e.g.*, 0), functions (*e.g.*, $\times$), and predicates (*e.g.*, $<$). The satisfiability of SMT formulae is interpreted within a *theory* of integers, real numbers, bit vectors, arrays, strings, *etc.*, which explains the origin of the name of SMT. The theory defines a variable domain (*e.g.*, the set of real numbers) and assigns a definite meaning to each constant/function/predicate symbol (*e.g.*, "$<$" being the lexicographical ordering on strings). For example, the formula $(x \times y < 12) \wedge (y - x = 3)$ is satisfiable within the theory of integer arithmetic, and $\langle x = 1, y = 4 \rangle$ is one solution.

Satisfiability modulo theories library (SMT-LIB) [4] is a prominent standard that provides rigorous specifications of background theories and an input/output language for expressing SMT formulae. The syntax of the SMT-LIB language is very similar to that of Lisp [20]. The following shows a portion of the SMT-LIB program for the above formula, whose expressions use prefix notation.

```
(set-logic QF_NIA)    ;quantifier-free integer arithmetic
(declare-const x Int) ;x is an integer variable
(declare-const y Int) ;y is an integer variable
(assert (and (< (* x y) 12) (= (- y x) 3)))
```

SMT solvers are software tools that deduce whether or not a set of formulae is satisfiable, among which Microsoft Z3 [11], STP [22], and Yices [16] are popular. Most SMT solvers accept an SMT-LIB program as input. To facilitate writing formulae, they may also provide APIs for other programming languages (*e.g.*, Python and Java). Because of the powerful functionality, SMT solvers have been adopted as an essential module for various tools across many application areas, such as software testing and program verification.

3 APPROACH AND IMPLEMENTATION

3.1 Problem Formulation

We summarize the syntax of types [34] in Figure 4. An *element type* is an atomic primitive type (*e.g.*, `float` for the set of 32-bit floating-point numbers). Currently, there are four allowable element types. A *tensor type* represents single- or multi-dimensional arrays whose elements have the same type. For convenience, we use $x.et$, $x.order$, $x.shape$, $x.fmt$ to denote the element type, order (*i.e.*, the number of dimensions), shape (*i.e.*, an array of all dimensional lengths), and format of a tensor variable $x : ts$, respectively. *Terms* are mathematical expressions built from constants and variables by applying n-place elementary operations [18]. Because binary operations such as addition, subtraction, multiplication, division, and modulo (denoted by mod) are mainly used, we specifically list their syntax rules. In most cases, refinement types use quantifier-free formulae, including Boolean constants, arithmetic comparison between two terms, and negation/conjunction/disjunction of existing formulae. Certain formulae may involve every dimension of a tensor variable whose order is not known in advance. Therefore, we add an implication rule with the universal quantifier $\forall v : b.\ p_1 \rightarrow p_2$, meaning that for each v of basic type b, if the condition p_1 holds, then so must p_2. However, the usage is restricted by bounded quantification [18] to ensure that the generated constraints remain decidable; that is to say, v is placed under an upper bound in the condition p_1 (*e.g.*, $v < x.order$). A *refinement* $\{v : p\}$ is a pair such that v is a variable and p is a logical formula. Types then include either *refined base types* (*i.e.*, basic types endowed with refinements) for tensors and hyperparameters or *dependent function types* [34] (*i.e.*, function types depending on parameter values) for DL operators. Note that the type of an operator with n inputs is written as $x_1 : \tau_1 \rightarrow \cdots \rightarrow x_n : \tau_n \rightarrow \mu$. A *context* is a sequence of variable-type bindings to record which variables are in scope.

The basic types are distinct from each other, so there is no subtype relationship [57] between any two refined base types with

$\langle$***Symbols***$\rangle$ sy ::= $0, -1, 1, \cdots \mid v, x, y, \cdots$ (constants/variables)
$\langle$***Element Types***$\rangle$ et ::= `bool` | `int` | `float` | `double`
$\langle$***Tensor Type***$\rangle$ ts ::= $et[sy] \mid ts[sy]$
$\langle$***Basic Types***$\rangle$ b ::= $et \mid ts$
$\langle$***Terms***$\rangle$ e ::= sy (constants and variables)
| $e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
| $e_1 \div e_2 \mid e_1 \bmod e_2 \mid \cdots$ (2-place expressions)
| $f\, e_1 \cdots e_n$ (n-place expressions)
$\langle$***Formulae***$\rangle$ p ::= `true`, `false` (Booleans)
| $e_1 = e_2 \mid e_1 \leq e_2 \mid \cdots$ (2-place predicates)
| $\neg p \mid p_1 \wedge p_2 \mid p_1 \vee p_2$ (negation/conjunction/disjunction)
| $\forall v : b.\ p_1 \rightarrow p_2$ (implication)
$\langle$***Refinements***$\rangle$ r ::= $\{v : p\}$ (known)
$\langle$***Types***$\rangle$ τ, μ ::= $b\{r\}$ (refined base)
| $v : \tau \rightarrow \mu$ (dependent function)
$\langle$***Context***$\rangle$ Γ ::= $\emptyset \mid \Gamma; v : \tau$ (variable-type binding)

Figure 4: Syntax of types.

different basic types. Supposing that p is a formula, $p[v := z]$ denotes that all the *free* (*i.e.*, not restricted by quantifiers) occurrences of v in p are substituted by z [18]. Subtyping (denoted by $<:$) on $b\{x : p_x\}$ and $b\{y : p_y\}$ is defined as follows [34]:

$$\Gamma \vdash b\{x : p_x\} <: b\{y : p_y\} \iff \Gamma \vdash \forall x : b.\ p_x \rightarrow p_y[y := x]\,. \quad \text{(Sub-Base)}$$

Therefore, `pos` is a subtype of `nat` (*i.e.*, `pos` $<:$ `nat`) because $\forall v :$ `int`. $0 < v \rightarrow 0 \leq v$ is a valid formula. Another example is that the 3×3 matrix type is a subtype of the 2-order, valid tensor type (which is the type of the first input parameter of `MatMul` in Figure 3).

As mentioned before, a DL model $\mathcal{M}$ is formally represented as the following directed acyclic computation graph [24]:

$$\mathcal{M} = \langle V = \{op_i\}_{i=1}^{N}, E = \{(op_i, op_j)\}_{i \neq j}, HP = \{hp_i\}_{i=1}^{K} \rangle\,.$$

Each node op_i is an operator specified by developers (*e.g.*, `Conv2d` and `AvgPool2d` in Figure 2). We ignore the automatically inserted operators for backward propagation because they are crafted by frameworks to calculate gradients in strict accordance with the forward operators. We assume that a backward operator should not introduce any type errors listed in Table 1 if its corresponding forward operator executes correctly. A directed edge (op_i, op_j) delivers a tensor from op_i to op_j and specifies that op_j must wait until the execution of op_i finishes. Each hyperparameter hp_i is endowed with a refinement type $b_i\{hp_i : p_i\}$ that defines the domain of hp_i (denoted by Δ_i). b_i is a basic type such as `int` or the basic tensor type, and p_i is a logical formula. For unified handling, the initial input tensors are treated as hyperparameters. We denote the *hyperparameter configuration space* of $\mathcal{M}$ as $\Delta = \Delta_1 \times \Delta_2 \times \cdots \times \Delta_K$. Each $\lambda \in \Delta$ is called a *hyperparameter vector*.

First, we explain the methodology of checking the validity of $\mathcal{M}$. Let $\Gamma = hp_1 : b_1\{hp_1 : p_1\}; \cdots; hp_K : b_K\{hp_K : p_K\}$ be the context of the model $\mathcal{M}$. For an operator op_i (similar to a conventional function call), we check for each input argument $v_{i,k}$ whether its type is a subtype of (denoted by $<:$) that of the corresponding formal input parameter $\overline{v_{i,k}}$. In other words, the context Γ must deduce the following judgment:

$$\Gamma \vdash b_{v_{i,k}}\{v_{i,k} : p_{v_{i,k}}\} <: b_{\overline{v_{i,k}}}\{\overline{v_{i,k}} : p_{\overline{v_{i,k}}}\}\,. \quad \text{(Chk-In)}$$

If so, we conclude that each output argument $y_{i,l}$ is associated with (denoted by $\mapsto$) the type of the corresponding formal output parameter $\overline{y_{i,l}}$ (with variable substitution):

$$\Gamma \vdash y_{i,l} \mapsto b_{\overline{y_{i,l}}}\{y_{i,l} : p_{\overline{y_{i,l}}}[\overline{y_{i,l}} := y_{i,l}]\}\,. \quad \text{(Chk-Out)}$$

For a directed edge (op_i, op_j) (similar to an assignment statement), we conclude that op_j's n-th input tensor $x_{j,n}$ is associated with the type of op_i's m-th output tensor $y_{i,m}$ (with variable substitution):

$$\Gamma \vdash x_{j,n} \mapsto b_{y_{i,m}}\{x_{j,n} : p_{y_{i,m}}[y_{i,m} := x_{j,n}]\}\,. \quad \text{(Chk-Edge)}$$

Next, we show how to generate a set of constraints whose satisfiability implies that $\mathcal{M}$ is valid. We observe that the algorithmic execution of $\mathcal{M}$ is represented as iterative forward and backward propagation on its computation graph. Model inference can be thought of as single-pass forward propagation. Let $S = \langle op_{i_1}, op_{i_2}, \cdots, op_{i_N} \rangle$ be the actual runtime execution ordering of operators, which is linearly extended from the edge ordering by referring to the framework implementations [41, 60]. We traverse the computation graph by strictly following S and generate the constraints according to the above three judgments. For the Chk-In judgment, we first generate $b_{v_{i,k}} = b_{\overline{v_{i,k}}}$ (*i.e.*, both basic types should be identical). We then generate $\forall v_{i,k} : b_{v_{i,k}}.\ p_{v_{i,k}} \rightarrow p_{\overline{v_{i,k}}}[\overline{v_{i,k}} := v_{i,k}]$ from the previous subtyping definition. However, since any defective hyperparameter vector leads to unsatisfiability, we eliminate the universal quantifier to discover all the defective vectors. We also notice that $p_{v_{i,k}}$ either is implied by Γ (if $v_{i,k}$ is a hyperparameter) or has been generated during the visit of a predecessor operator. Therefore, we finally use the simplified constraint $p_{\overline{v_{i,k}}}[\overline{v_{i,k}} := v_{i,k}]$. For the Chk-Out judgment, we generate $b_{y_{i,l}} = b_{\overline{y_{i,l}}}$ and $p_{\overline{y_{i,l}}}[\overline{y_{i,l}} := y_{i,l}]$. For the Chk-Edge judgment, we generate a simpler constraint $x_{j,n} = y_{i,m}$.

Finally, we formulate the problem of checking the validity of $\mathcal{M}$ as a constraint satisfaction problem (CSP) [64]. Being a general and useful concept, CSP abstracts a problem as finding a solution to a set of imposed constraints that must be satisfied as conditions by the variables. It is an important research subject in artificial intelligence (AI). Well-known CSPs include eight queens puzzle and graph coloring. Our problem is defined as a triple $\langle V, D, C \rangle$ [64]:

$$\begin{aligned} V &= \{V_1, V_2, \cdots, V_K \mid V_i = hp_i \text{ for } i \in [1, K]\}\,, \\ D &= \{D_1, D_2, \cdots, D_K \mid D_i = \Delta_i \text{ for } i \in [1, K]\}\,, \\ C &= (\textstyle\bigcup_{i=1}^{N} C_i) \cup \{x_{j,n} = y_{i,m} \mid (op_i, op_j) \in E\}\,, \\ C_i &= \{b_{v_{i,k}} = b_{\overline{v_{i,k}}},\ \ p_{\overline{v_{i,k}}}[\overline{v_{i,k}} := v_{i,k}]\} \\ &\quad \cup \{b_{y_{i,l}} = b_{\overline{y_{i,l}}},\ \ p_{\overline{y_{i,l}}}[\overline{y_{i,l}} := y_{i,l}]\}\,. \end{aligned}$$

V is a set of variables that are actually the hyperparameters. D represents the respective variable domains such that each D_i is the domain of hp_i. C contains the generated constraints mentioned above. For a hyperparameter vector $\lambda = \langle v_1, v_2, \cdots, v_K \rangle \in \Delta$, $\mathcal{M}$ is valid with λ if the above set C is satisfiable when each $V_{i \in [1,K]}$ is assigned the corresponding v_i; otherwise, λ results in type errors.

3.2 Refinement Types of DL Operators

In this section, we take `Conv2d`, the 2D convolution operator, as an example to demonstrate how to define refinement types. We consider the following hyperparameters: number of input channels (*ic*), number of output channels (*oc*), tensor format (*fmt*), kernel size (*ks*), stride (*sd*), padding (*pad*), dilation (*dil*), and number of groups

$$\texttt{Conv2d} :: x : ts\{x : p_{x\text{-}1} \wedge p_{x\text{-}2}\} \to ic : \texttt{pos} \to oc : \texttt{pos} \to fmt : \{\texttt{NCHW}, \texttt{NHWC}\} \to ks : \texttt{pos}[2] \to pad : \texttt{pos}[2] \to dil : \texttt{pos}[2]$$
$$\to sd : \texttt{pos}[2]\{sd : p_{sd\text{-}1} \wedge p_{sd\text{-}2}\} \to gp : \texttt{pos}\{gp : p_{gp}\} \to ts\{y : p_{y\text{-}1} \wedge p_{y\text{-}2} \wedge p_{y\text{-}3}\}$$

$$p_{x\text{-}1} \doteq (x.order = 4) \wedge (0 < x.shape[0]) \wedge (0 < x.shape[1]) \wedge (0 < x.shape[2]) \wedge (0 < x.shape[3]) \wedge ((x.et = \texttt{float}) \vee (x.et = \texttt{double}))$$

$$p_{x\text{-}2} \doteq (x.fmt = fmt) \wedge (\neg IsPT \vee (x.fmt = \texttt{NCHW})) \wedge (x.shape[\texttt{CHANNEL}] = ic)$$

$$p_{sd\text{-}1} \doteq (sd[0] \leq ks[0]) \wedge (sd[1] \leq ks[1])$$

$$p_{sd\text{-}2} \doteq \neg IsTF \vee ((dil[0] = 1) \wedge (dil[1] = 1)) \vee ((sd[0] = 1) \wedge (sd[1] = 1))$$

$$p_{gp} \doteq ((ic \bmod gp) = 0) \wedge ((oc \bmod gp) = 0)$$

$$p_{y\text{-}1} \doteq (y.et = x.et) \wedge (y.order = x.order) \wedge (y.fmt = x.fmt) \wedge (y.shape[0] = x.shape[0]) \wedge (y.shape[\texttt{CHANNEL}] = oc)$$

$$p_{y\text{-}2} \doteq (y.shape[\texttt{HEIGHT}] = ((x.shape[\texttt{HEIGHT}] + 2 \times pad[0] - dil[0] \times (ks[0] - 1) - 1) \div sd[0] + 1))$$
$$\wedge (y.shape[\texttt{WIDTH}] = ((x.shape[\texttt{WIDTH}] + 2 \times pad[1] - dil[1] \times (ks[1] - 1) - 1) \div sd[1] + 1))$$

$$p_{y\text{-}3} \doteq (0 < y.shape[0]) \wedge (0 < y.shape[1]) \wedge (0 < y.shape[2]) \wedge (0 < y.shape[3])$$

Figure 5: Refinement type of the `Conv2d` operator. Tunable hyperparameters include the numbers of input (*ic*) and output (*oc*) channels, tensor format (*fmt*), kernel size (*ks*), stride (*sd*), padding (*pad*), dilation (*dil*), and number of groups (*gp*). `pos` refers to the refinement type of positive integers, and *ts* is the basic tensor type. To save space, `CHANNEL`, `HEIGHT`, and `WIDTH` denote the indices of the channel, height, and width dimensions, respectively. As usual, *x.et*, *x.order*, *x.shape*, *x.fmt* denote the element type, order, shape, and format of the tensor variable *x*, respectively.

(*gp*). The tensor format is either `NCHW` or `NHWC`; other hyperparameters require positive integer values, although their declarations use the Python `int` type. We assume that each of the kernel size, stride, padding, and dilation is an array of two positive integers that are used for the height and width dimensions, respectively.

The core computation of `Conv2d` is described as follows [61]:

$$y(i, j) = \text{bias}(j) + \sum_{k=0}^{ic-1} \text{weight}(j, k) \star x(i, k)\,,$$

where $0 \leq i <$ the batch size, $0 \leq j < oc$, x and y are the input and output tensors, and $\star$ is the 2D cross-correlation [80] operation. From the mathematical definition [24] and framework implementations of `Conv2d`, we extract the following computational constraints on both tensors and hyperparameters and show the formalized refinement type of `Conv2d` in Figure 5, where item labels correspond to the names of the logical formulae.

- $p_{x\text{-}1}$ The input tensor should have exactly four dimensions whose lengths are positive integers. Its elements are floating-point numbers, *i.e.*, the element type is either `float` or `double`.
- $p_{x\text{-}2}$ The actual format of the input tensor should be equal to the tensor format hyperparameter; for PyTorch models, it must be `NCHW`. The latter is a PyTorch-specific constraint, so we use a Boolean control variable *IsPT* initialized to `true` when the framework is PyTorch. The length of the input tensor's channel dimension should be equal to the number of input channels. Note that the dimensional indices except that of the batch dimension (always 0) are not constants because of two possible tensor formats. To save space in Figure 5, we use `CHANNEL`, `HEIGHT`, and `WIDTH` to denote the indices of the channel, height, and width dimensions, respectively.
- $p_{sd\text{-}1}$ Each element of the stride cannot exceed that of the kernel size; otherwise, some input data is mistakenly skipped.
- $p_{sd\text{-}2}$ TensorFlow additionally requires that the stride and dilation cannot both be greater than 1 [74]. Similarly, we use another Boolean control variable *IsTF* in this constraint, which is initialized to `true` when the framework is TensorFlow.
- p_{gp} The number of groups specifies the "number of blocked connections from input channels to output channels" [61]; therefore, the numbers of input and output channels should both be divisible by it.
- $p_{y\text{-}1}$ The output tensor has the same element type, order, format, and length of the batch dimension with the input tensor. The length of its channel dimension should be equal to the number of output channels.
- $p_{y\text{-}2}$ The output height and width are calculated as follows [61]:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times pad[0] - dil[0] \times (ks[0] - 1) - 1}{sd[0]} + 1 \right\rfloor,$$
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times pad[1] - dil[1] \times (ks[1] - 1) - 1}{sd[1]} + 1 \right\rfloor.$$

  Both PyTorch and TensorFlow also allow developers to pass a string "valid" or "same" as the value of the padding hyperparameter. "valid" means no padding. "same" means that the framework automatically pads the input evenly; therefore, the above calculation can be simplified [75] as $H_{out} = \lceil \frac{H_{in}}{sd[0]} \rceil$ and $W_{out} = \lceil \frac{W_{in}}{sd[1]} \rceil$. When using the "same" padding, PyTorch requires that both elements of the stride must be 1, so the output has the same height and width as the input.
- $p_{y\text{-}3}$ The output tensor is also legal; that is to say, each length of the four dimensions is a positive integer.

At present, we support 70+ types of DL operators. Refty is extensible to incorporate new operators, which is discussed in Section 5.1.

3.3 Workflow

Figure 6 illustrates how Refty works. Refty accepts a computation graph, a model specification, and a tool specification as input from developers or AutoML tools. The computation graph is reconstructed from a serialized PyTorch or TensorFlow model in the form of disk files, using our front-end parsers that invoke the framework built-in model deserialization APIs. Also, developers can describe

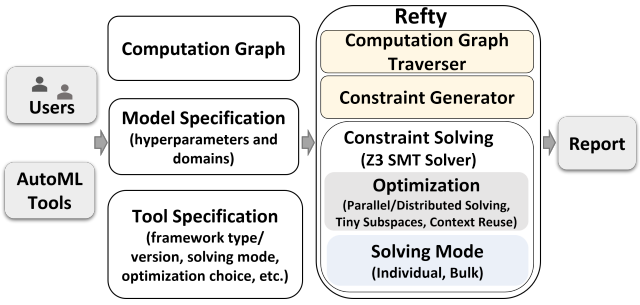

Figure 6: Workflow of Refty.

such a graph manually in the Protocol Buffers [25] language, using the intermediate representation of MMdnn [44], whose syntax is very similar to that of ONNX [53]. The model specification includes the tunable hyperparameters and their domain definitions. Unspecified hyperparameters will use the framework default values. The tool specification contains the framework type/version, solving mode, optimization choice, *etc.*

Refty traverses the computation graph by following the operator execution ordering to generate a set of constraints that the DL model should satisfy. It then utilizes Microsoft Z3 [11] to obtain the unsatisfiable hyperparameter vectors that result in type errors. Refty chooses SMT solvers for two reasons: (1) the constraints can be specified with SMT solvers' built-in types (*e.g.*, integer and array), algebraic data types (*e.g.*, scalar and record), functions (*e.g.*, multiplication and modulo), and value comparison (*e.g.*, equality and less-than); (2) they handle nonlinear (*e.g.*, calculating the output size of `Conv2d`) and higher-order mathematical expressions very efficiently. Nevertheless, the scalability of SMT solvers may potentially limit the usability of Refty, which is discussed in Section 5.2. In the final report to users, Refty presents the corresponding hyperparameter vector for each detected error. If needed for further diagnosis, Refty additionally reports all the error-related constraints and the operators to which these constraints belong.

3.4 Graph Traversal and Constraint Generation

For each supported type of operator, Refty aforehand translates the logical formulae of its refinement type to SMT-LIB [4] statements in a reusable template, using the Python API of Z3. The symbol names of tensors and hyperparameters are parameterized to handle variable substitution. Refty declares integer constants for the available element types and known tensor formats. To denote the tensor type, it defines a custom record type with four fields, representing the element type, format, order (an integer), and shape (an array) of the tensor, respectively. The manipulation over all the dimensions of a tensor with a non-constant order (*e.g.*, calculating the total number of tensor elements) is implemented by recursive functions.

Refty first creates a Z3 instance, checks the basic types of the hyperparameters specified in the model specification, and defines all the hyperparameter symbols. It then traverses the computation graph one operator by another by following a predefined sequence $S = \langle op_{i_1}, op_{i_2}, \cdots, op_{i_N} \rangle$ that denotes the actual runtime execution ordering of operators. S is linearly extended from the edge ordering by reference to the framework implementations [41, 60]. When visiting an operator op, Refty locates the SMT-LIB template by op's type, checks the basic types of both input and output arguments against those of the corresponding formal parameters, fills in the actual symbol names derived from op's name, and executes the SMT-LIB statements to add constraint formulae to the solver instance. Refty also inserts equality constraints that assert that op's input tensor arguments are exactly the same as those outputs of the immediate predecessors. After the graph traversal finishes, Refty finally adds multiple formulae defining the domain of each hyperparameter.

3.5 Constraint Solving

Refty utilizes Microsoft Z3 to solve the generated constraints. To reduce the nondeterminism in Z3's conflict-driven clause learning (CDCL) [68] implementation, we split a constraint formula into as many smaller, self-contained ones as possible and add them one by one to the solver instance in our SMT-LIB templates. If users need to diagnose the root cause of an error, Refty employs the *minimal unsatisfiable core* (*i.e.*, an unsatisfiable subset containing the least number of the original formulae) returned by Z3 to report all the error-related constraints and the operators to which these constraints belong.

Refty implements two solving modes: *individual* mode and *bulk* mode. Under the individual mode, Refty independently solves the constraints for each vector in the hyperparameter configuration space. This mode is simple, requires less effort, and facilitates tool integration. For example, AutoML tools can work nearly as usual: they need to invoke Refty with the hyperparameter vector being explored just before launching a trial job. However, there exist continuous, significant warm-up overheads of traversing the computation graph, generating the constraints, and initializing the solver instance. Refty applies the *context reuse* technique to improve solving performance: it reuses the existing Z3 instance and almost all the initialized constraint formulae, pops out the domain definition formulae that were added last, and pushes new domain definition formulae for the next hyperparameter vector.

Under the bulk mode, Refty solves the constraints for once with the whole hyperparameter configuration space. However, Z3 returns only one satisfiable hyperparameter vector when there is any, instead of supplying all at once. Therefore, Refty implements a simple *AllSAT* (all solutions SAT) [76] feature: it iteratively invokes the same Z3 instance by appending the negation of the newfound solution to derive the next one. Supposing that $\lambda = \langle v_1, v_2, \cdots, v_K \rangle$ is a newfound satisfiable vector, Refty will append the constraint $(hp_1 \neq v_1) \wedge (hp_2 \neq v_2) \wedge \cdots \wedge (hp_K \neq v_K)$. As more and more solutions are discovered, AllSAT may get slower and slower because the appended negation formulae are becoming too many. After obtaining the set of the satisfiable hyperparameter vectors, Refty uses the set difference operation to derive all the unsatisfiable ones. Finally, it reruns completely from the beginning under the individual mode for each unsatisfiable hyperparameter vector to get the minimal unsatisfiable cores.

Constraint solving may slow down significantly if there exist a large number of constraints or a huge hyperparameter configuration space. Refty adopts two optimization techniques proposed by

DnnSAT [23]. The first is parallel (using worker threads) and distributed (using Spark [83]) solving, which divides the solving task into subtasks and processes them simultaneously. The other is tiny subspaces: the original hyperparameter configuration space is divided into numerous, disjoint, small subspaces to reduce the AllSAT overhead and tackle the skew problem [3]. These two techniques can be used in conjunction with context reuse.

4 EVALUATION

4.1 Experimental Design

Refty is evaluated on individual operators and real-world models under two mainstream DL frameworks of PyTorch (v1.5.1) and TensorFlow (v1.13.1). We aim to answer the following research questions (RQs):

RQ1: How effective is Refty on individual DL operators?
RQ2: How effective is Refty on real-world DL models?
RQ3: How efficient is Refty in constraint solving?
RQ4: How does Refty compare with related work?

We fine-tune several hyperparameters that are often tuned by developers in practice, including the batch size, input image size ("height × width" for convenience), tensor format, kernel size, stride, number of features, *etc.* For the hyperparameter values, we first select some typical ones, such as 8 and 16 [36] for the batch size, `NHWC` and `NCHW` for the tensor format, 1×1, 3×3, and 5×5 for the kernel size of `Conv2d` [69, 71], 2×2 for the kernel size of `MaxPool2d` [69, 71], 1×1 and 2×2 for the stride [69, 71], and 256 [63] for the number of features. The initial input size for the ImageNet dataset [13] is usually 224×224, while Inception-V3 uses 299×299 by default. Some neighboring values (*e.g.*, 4×4 for the kernel size and stride) are also selected. We further try a few smaller (*e.g.*, 32 for the number of features), larger (*e.g.*, 12×12 for the kernel size), or even invalid values that may lead to potential errors.

In order to obtain the ground truth of whether or not a hyperparameter vector λ results in type errors, we feed λ to the model training program and run it with one batch of input data. Assertions are also added to the program to detect errors that may not fail the execution but cause the final model to produce wrong output results (*e.g.*, an input tensor having an unsupported format). If the program crashes, we say that λ is an actual positive; otherwise, λ is an actual negative. We then run Refty on the hyperparameter configuration space of the model, compare each result (including the hyperparameter vector and possible root cause) with the ground truth, and calculate the numbers of true positives, false positives (*i.e.*, correct hyperparameter vectors being misreported as defective), true negatives, and false negatives (*i.e.*, defective hyperparameter vectors not being detected).

We adopt the standard metrics of *Precision* and *Recall* to assess the effectiveness of Refty, which are defined as follows [51]:

$$\text{Precision} = \frac{tp}{tp + fp} \times 100\%, \qquad \text{Recall} = \frac{tp}{tp + fn} \times 100\%.$$

The above *tp*, *fp*, *tn*, and *fn* denote the numbers of true positives, false positives, true negatives, and false negatives, respectively. Consequently, the numbers of actual positives and actual negatives are equal to $tp + fn$ and $fp + tn$, respectively. The higher Precision and Recall values, the more effective Refty is.

We conduct the experiments on an Azure Standard ND12 virtual machine [48] that has 12 Intel Xeon E5-2690V3 vCPUs (2.60 GHz, 30M Cache) and 112 GB main memory, running Ubuntu 18.04.

4.2 RQ1: How effective is Refty on individual DL operators?

We select five typical operators of `Conv2d`, `MaxPool2d` (2D max pooling), `Linear`, `Add` (addition of two tensors), and `Concat` (concatenation of a list of tensors in a given dimension) to evaluate the effectiveness of Refty.

For `Conv2d` and `MaxPool2d`, we tune their batch size (8 and 16), tensor format (`NCHW` and `NHWC`), number of input channels (3), input image size (7×7, 14×14, 27×27, and 224×224), kernel size (3×3, 6×6, 9×9, 12×12, and 15×15), and stride (1×1, 2×2, 4×4, and 8×8, 10×10).

For `Linear`, we use a 4-order image tensor as input and tune its batch size (8, 16, 24, and 32), number of input channels (3), and input image size (7×7, 14×14, 27×27, and 224×224). We also tune the numbers of input features (147, 588, 2187, and 150528) and output features (−128, 128, 256, 512, and 1024) of `Linear`.

For `Add`, we use two 4-order image tensors and independently tune their batch size (8 and 16), input image size (7×7, 14×14, 27×27, and 224×224), and number of input channels (3, 6, and 9).

For `Concat`, the input sequence consists of two 4-order image tensors. We independently tune their element type (`float` and `bool`), batch size (8), input image size (7×7, 14×14, 27×27, and 224×224), and number of input channels (3, 6, and 9).

We devise the above hyperparameter domains to cover all the error categories shown in Table 1. For example, when the input image size is 7×7 and the kernel size is 15×15 in the `Conv2d` experiments, Illegal Shape errors are raised. We also use a negative value as the number of output features of `Linear`, which leads to Illegal Value errors. To obtain the ground truth, we compose a program for training a model that contains only the experimental operator. Because of the small hyperparameter configuration spaces, Refty runs under the individual solving mode for simplicity. The average warm-up time of Refty is 0.04 second for `Conv2d`, `MaxPool2d`, and `Linear` and is 0.1 second for `Add` and `Concat`. The solving time is 0.012 second per hyperparameter vector on average.

Table 2 shows the experimental results. Refty finds all the type errors and does not introduce any false alarms. It achieves 100% Precision and Recall, which demonstrates the effectiveness of our formulated refinement types of DL operators.

4.3 RQ2: How effective is Refty on real-world DL models?

We evaluate Refty on five real-world DL models: AlexNet [39], VGG-16 [69], Inception-V3 [71], LSTM [29]-based Seq2Seq [70], and GRU [8]-based Seq2Seq. They are representative in the areas of computer vision, natural language processing, and speech recognition. Some of them, such as LSTM and GRU, are also widely used in various DL for software engineering applications [26, 31, 43, 84].

For AlexNet and VGG-16, we set the number of input channels to 3, input batch size to 8 or 16, and input image size to 224×224. We tune the kernel size (3×3, 6×6, 9×9, and 12×12) and stride (1×1, 2×2, 4×4, and 8×8) of the first `Conv2d` operator. We also

Table 2: Experimental results on individual operators.

Metrics \ Operator	Conv2d	MaxPool2d	Linear	Add	Concat
PyTorch Total	400	400	320	576	576
True Positive	240	230	256	552	540
True Negative	160	170	64	24	36
False Positive	0	0	0	0	0
False Negative	0	0	0	0	0
Precision	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%
TensorFlow Total	400	400	320	576	576
True Positive	240	230	256	552	540
True Negative	160	170	64	24	36
False Positive	0	0	0	0	0
False Negative	0	0	0	0	0
Precision	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%

Table 3: Experimental results on real-world DL models.

Metrics \ Model	AlexNet	VGG-16	Inception-V3	Seq2Seq (LSTM)	Seq2Seq (GRU)
PyTorch Total	512	512	256	256	256
True Positive	200	200	197	240	240
True Negative	312	312	59	16	16
False Positive	0	0	0	0	0
False Negative	0	0	0	0	0
Precision	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%
TensorFlow Total	512	512	256	256	256
True Positive	200	320	57	240	240
True Negative	312	192	199	16	16
False Positive	0	0	0	0	0
False Negative	0	0	0	0	0
Precision	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%

try different kernel sizes (1×1, 2×2, 4×4, and 8×8) and strides (1×1, 2×2, 4×4, and 8×8) for the first `MaxPool2d` operator. For Inception-V3, we use identical hyperparameter domains except that the input batch size is 8 and the input image size is 299×299.

For the other two Seq2Seq models, we tune the number of features in the hidden state (32, 64, 128, and 256) and the number of recurrent layers (32, 64, 128, and 256) for both encoder and decoder.

We implement AlexNet by ourselves, download the training programs and serialized model files of VGG-16 and Inception-V3 from the official websites [62, 82], and implement Seq2Seq by referring to open-source code [19, 77]. It takes about 6 to 7 seconds to run the training programs with one batch of input data and one hyperparameter vector for obtaining the ground truth, excluding the time of data preparation and model validation/testing. Refty still runs under the individual solving mode. The numbers of Z3 symbols and constraint formulae of AlexNet, VGG-16, Inception-V3, LSTM-based Seq2Seq, and GRU-based Seq2Seq are 112/183, 196/345, 920/1788, 16/28, and 16/28, respectively. The average warm-up time and solving time (in seconds) of Refty are 0.5/0.029, 0.67/0.045, 3.17/0.18, 0.11/0.013, and 0.11/0.013, respectively.

Table 3 shows the experimental results. Refty again achieves 100% Precision and Recall in all the experiments, which demonstrates its effectiveness on DL models. We notice a discrepancy in the ground truth of the VGG-16 and Inception-V3 experiments. The root cause lies in the differences between the official training programs. In the TensorFlow VGG-16 experiments, many hyperparameter vectors cause the output shape of the last `MaxPool2d` to violate the requirement of the succeeding `Linear`, which brings 120 more crashes. Instead, the PyTorch program inserts an extra `AdaptiveAvgPool2d` (2D adaptive average pooling) operator with proper padding to resolve this issue. In the PyTorch Inception-V3 experiments, the fixed padding values make the PyTorch program more vulnerable to Illegal Shape errors and lead to 140 more crashes.

4.4 RQ3: How efficient is Refty in constraint solving?

We evaluate Refty on the constraint solving efficiency with the optimization techniques mentioned in Section 3.5. Inception-V3 [71] is selected as our experimental object; it is one of the representative real-world models in computer vision and has a more complex neural architecture (in terms of width and height) than the other four DL models. The tunable hyperparameters include the input batch size (16 and 32), input image size ($\{299, 300, 301\} \times \{299, 300, 301\}$; 9 different sizes), and kernel size ($\{1, 3, 5\} \times \{1, 3, 5\}$; 9 different sizes) for the first three `Conv2d` operators. Therefore, the hyperparameter configuration space of Inception-V3 consists of 13,122 vectors in total. To increase the solving difficulty, we carefully devise the above hyperparameter domains to make every hyperparameter vector satisfy the constraints.

For parallel solving, we create 1, 4, 8, and 12 worker threads. More threads are not considered since there are only 12 vCPUs on the experimental machine. For tiny subspaces, we try subspace sizes of equipartition, 200, 100, and 50. "Equipartition" means that the original hyperparameter configuration space is simply divided equally by the number of threads. For example, if we use 12 worker threads, each will independently solve one subspace that includes about $\lceil \frac{13,122}{12} \rceil = 1{,}094$ hyperparameter vectors. Context reuse is enabled by default for all the experiments. The reason is that the overhead of a complete Z3 instance initialization is evident (3.17 seconds on average, as reported in Section 4.3) and causes some experiments to run too long.

Table 4 demonstrates the end-to-end execution time (in seconds) of Refty under both individual and bulk solving modes. We also show the speedup relative to the baseline experiment (individual mode: 1 thread; bulk mode: 1 thread + equipartition). Our results confirm the strong effects of the optimization techniques. Under the individual mode, parallel solving achieves a near-linear speedup from 3.9X to 11.8X since the computational complexity of solving each hyperparameter vector is identical. Under the bulk mode, the overall speedup ranges from 3.5X to 51.0X. Simply creating more worker threads (see the "Equipartition" row) achieves a super-linear speedup from 10.3X to 43.4X because smaller hyperparameter configuration spaces observably reduce the overhead of AllSAT. Tiny subspaces (see the "1 thread" column) are also effective for the above same reason. After combining these two techniques, Refty achieves a more promising and near-linear speedup from 10.3X to

Table 4: Runtime performance of Refty with parallel solving and tiny subspaces. Context reuse is enabled by default.

Mode	Size of Subspace	Number of Threads			
		1	4	8	12
Individual	1	1.0 (2,365.3 s)	3.9 (594.9 s)	7.9 (298.8 s)	11.8 (200.4 s)
Bulk	Equipartition	1.0 (8,660.1 s)	10.3 (835.3 s)	28.3 (305.5 s)	43.4 (199.4 s)
	200	3.5 (2,450.8 s)	14.0 (615.4 s)	27.9 (309.5 s)	41.7 (207.5 s)
	100	3.92 (2,208.9 s)	15.6 (554.7 s)	31.0 (279.1 s)	46.2 (187.3 s)
	50	4.3 (1,998.1 s)	17.2 (502.3 s)	34.2 (252.9 s)	51.0 (169.8 s)

Note: The two values in a cell represent the speedup and execution time in seconds, respectively.

51.0X. As the subspace size decreases, the overhead of AllSAT also continues to reduce, and the performance gain gradually increases.

4.5 RQ4: How does Refty compare with related work?

We compare Refty with Pythia [40], a static shape-checking tool for TensorFlow Python programs. Pythia detects incompatible shapes by modeling and tracking tensor shapes across TensorFlow API calls. The analysis of Pythia uses the WALA [79] framework and declarative Datalog [5] rules. We select the Unaligned-Tensor-1 (UT-1) program[1] as the experimental object because Pythia does not support the TensorFlow Slim API [66] used by the training programs of our evaluated real-world models (Section 4.3). UT-1 is one of the fourteen programs provided by Zhang *et al.* [87] that contain Unaligned Tensor bugs, and it has been successfully analyzed by Pythia. UT-1 trains a convolutional model for MNIST handwritten digit classification [14], which is much more complicated than those implemented in other UT programs.

Since Pythia checks tensor shape mismatches, we devise the hyperparameter configuration space such that only Illegal Shape and Incompatible Shape errors, if any, may be raised. Before the experiment, we have actually tried some hyperparameter vectors that led to errors in the other four categories and noticed that Pythia could not detect them. UT-1 invokes the Primitive Neural Net API [73] (*e.g.*, `tf.nn.conv2d`) that uses a new *filter* hyperparameter instead of the kernel size for convolutional operators. The filter is a 4-order tensor of the shape [kernel height, kernel width, number of input channels, number of output channels]. We randomly select 9 and 10 shapes (such as "[5, 5, 1, 32]" and "[33, 33, 3, 64]") for the filter hyperparameter of the two `Conv2d` operators, respectively. We also tune the kernel size (2×2, 12×12, and 22×22) of the first `MaxPool2d` operator and set the padding of the two `MaxPool2d` operators to "valid". Therefore, there are 270 hyperparameter vectors in total. For each vector, we fill the corresponding hyperparameter values in the UT-1 program and then pass the program file to Pythia for analysis. Pythia starts with a ten-minute warm-up, and after that, each checking takes 12 seconds on average. Refty runs under the individual solving mode, and its warm-up time and solving time are about 0.037 second and 0.019 second per vector, respectively.

[1] https://github.com/ForeverZyh/TensorFlow-Program-Bugs/tree/master/StackOverflow/UT-1/38167455-buggy/mnist.py

Table 5: Comparison with Pythia.

Metrics \ Tool	Pythia	Refty
TensorFlow Total	270	270
True Positive	33	270
True Negative	0	0
False Positive	0	0
False Negative	237	0
Precision	100%	100%
Recall	12.2%	100%

Table 5 shows the experimental results. Refty still achieves 100% Precision and Recall. Although the Precision of Pythia is 100%, its Recall is only 12.2%, a rather low percentage. The reason is that Pythia does not detect 237 buggy cases. In 153 of them, a tensor of an illegal shape is produced by a `Conv2d` operator. In the other 84 cases, the input and filter tensors of the first `Conv2d` operator do not have a matched number of input channels, which causes Incompatible Shape errors.

5 DISCUSSION

5.1 Extensibility of Refty

Currently, Refty supports two mainstream DL frameworks (PyTorch and TensorFlow) and 70+ types of commonly used operators. Refty can be adapted to other frameworks such as Apache MXNet [7] and ONNX [53] because they also adopt the same abstraction to represent models as tensor-oriented computation graphs. The syntax and semantics of their graphs and operators are very similar to those of PyTorch and TensorFlow; therefore, existing SMT-LIB templates of the refinement types can be reused with minor modifications. Nevertheless, the graph traversal may need to be adjusted to accord with the actual operator execution ordering of other frameworks. Refty is also extensible to incorporate new DL operators. To support a new type of operator, users need to extract the computational constraints, formulate a refinement type for each formal input/output parameter based on the operator's mathematical definition and framework implementations, and implement an SMT-LIB program template.

5.2 Threats to Validity

We discuss the following main threats to the validity of our work.

Refinement types. We formulate the refinement types of DL operators by our domain experience based on their mathematical definitions. We also examine the source code of both PyTorch and TensorFlow to incorporate framework-specific computational constraints into the formulation. Nevertheless, because of the large manual effort involved in formulating refinement types and preparing SMT-LIB templates, there might be a potential inaccuracy. To mitigate this threat, we strived to reach group consensus before making decisions, and we continuously refined our approach by carefully referring to the documentation and framework source

code. In our experiments, Refty achieves 100% Precision and Recall, confirming the validity of the refinement types of DL operators.

SMT solving. Refty utilizes Microsoft Z3 to solve the generated constraints. As DL models become more sophisticated and hyperparameter configuration spaces enlarge gradually, the computational complexity of constraint solving will increase observably and thus may reduce the usability of our tool. Currently, Refty implements context reuse and adopts another two optimization techniques of parallel/distributed solving and tiny subspaces from DnnSAT [23] to accelerate constraint solving. Our experiments on a fairly large hyperparameter configuration space with 13,122 vectors have demonstrated the strong effects of these optimizations (Table 4). In the future, we may advance SMT solvers and try larger-scale distributed solving to better tackle this problem.

6 RELATED WORK

There are some recent empirical studies on deep learning program defects and job failures. Zhang *et al.* [87] studied 175 TensorFlow program bugs collected from Stack Overflow pages and GitHub commits, 24 (about 13.71%) of which were caused by unaligned tensors. Islam *et al.* [30] extended this work to cover more frameworks such as Torch [10] and Caffe [35]; they discovered that the same reason was a major root cause (about 13.48%) of the total 415 bugs and was the leading cause of Torch bugs. Zhang *et al.* [85] conducted an empirical study on 4,960 failed DL jobs collected from an internal platform of Microsoft; they found that about 57 (8.53%) out of the 668 DL-specific failures were caused by mismatched tensors. Zhang *et al.* [86] examined 715 Stack Overflow questions to study common challenges in DL application development, many of which asked for help on shape inconsistency bugs. These empirical studies indicate that type errors of DL models are not uncommon and motivate us for a formal and systematic solution.

There have been many type-based techniques to perform correctness checking on programs. Some works [6, 17, 27, 32, 33, 50, 67] enrich the type system with specific tensor properties for static verification. For example, Eaton [17] proposed strongly typed linear algebra for numerical algorithms in which the dimensions of matrices were exposed. A few others [2, 37, 45, 81] reason whether the array index is out of bounds. For instance, Index Checker [37] is an open-source tool to discover array access errors in Java programs whose type system is conditioned by cooperating hierarchies of dependent types. As described before, these previous works target programs written in conventional programming languages (*e.g.*, Haskell and Java) and cannot be applied to DL models directly. Recently, Wen *et al.* [38] leveraged refinement types and constrained the values of tensor elements to verify the adversarial robustness of neural networks, which was "commonly characterised as the deviation in the neural network's outputs given perturbations of its inputs" [38]. Two libraries implemented in F* and Liquid Haskell were provided for constructing fully connected neural networks that use four common activation functions. To make verification tractable, users need to reduce the tensor dimensionality and model size manually. In comparison, Refty targets a different problem and supports more types of DL operators: it constrains hyperparameter values and tensor shapes/element types/formats to check the validity of models and eliminate common type errors. Leonardo *et al.* [58] proposed TensorSafe, a Haskell library that helps developers define the neural architectures of DL models and leverages the Haskell type system to validate the structural correctness. TensorSafe does not use refinement types to describe further computational constraints on tensors and hyperparameters. Therefore, although TensorSafe detects Incompatible Shape errors, it cannot find errors in the other categories shown in Table 1.

Popular DL frameworks adopt a hybrid programming paradigm: the runtime is implemented in C++ for high execution performance, while developers use other language bindings (*e.g.*, Python) for flexible programmability. Some recent works try to integrate DL frameworks directly into programming languages. Swift for TensorFlow [65] (which was archived in February 2021) implements language-integrated differentiable programming and allows developers to construct models with the Swift language and the familiar TensorFlow API. ONNX-Scala [46] brings full support for ONNX to the Scala ecosystem. Powered by the type systems of Swift and Scala, the two works are able to detect some type errors of DL models at compile-time. For example, Swift for TensorFlow checks whether the types of hyperparameter arguments are expected, but it does not inspect hyperparameter values and track tensor shapes. Furthermore, they only support a specific framework and cannot be adapted to others.

Recently, researchers proposed static and dynamic techniques to detect bugs in DL programs. Ariadne [15] applies WALA [79] program analysis to TensorFlow Python code and tracks tensors via a custom type system. Pythia [40] is a static shape-checking tool for TensorFlow programs that uses WALA and declarative Datalog [5] rules. Pythia models and tracks tensor shapes across TensorFlow API calls and checks whether incompatible shapes exist. ShapeFlow [78] modifies the implementation of TensorFlow API to capture and manipulate tensor shapes and runs DL programs to detect shape incompatibility issues. These tools are friendly to developers since they directly operate on Python code. However, they are TensorFlow-specific, focus on Incompatible Shape errors, and have difficulty detecting other errors caused by improper hyperparameter values, unsupported tensor formats, *etc.* Refty is a novel refinement type-based tool for both PyTorch and TensorFlow. It rigidly formulates the computational constraints enforced by operators and systematically detects invalid DL models caused by six categories of type errors. Refty also achieves good runtime performance; for example, it checks 13,122 hyperparameter vectors of Inception-V3 in about 200 seconds.

7 CONCLUSION

In this paper, we have presented Refty, a refinement type-based tool for statically detecting common type errors of deep learning models. These errors are caused by tensors and hyperparameters that violate the computational constraints. The problem of checking the validity of a model is formulated as a constraint satisfaction problem, and Refty utilizes an SMT solver for solving the constraints. Our experiments demonstrate that Refty is very effective at detecting potential type errors before job execution.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard,

et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) *(SSYM '09)*. USENIX Association, USA, 51–66.

[3] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. 2011. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 287–300. https://doi.org/10.1145/1966445.1966472

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[5] S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1 (mar 1989), 146–166. https://doi.org/10.1109/69.43410

[6] Tongfei Chen. 2017. Typesafe Abstractions for Tensor Operations (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 45–50. https://doi.org/10.1145/3136000.3136001

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[8] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[9] François Chollet. 2017. *Deep Learning with Python*. Manning.

[10] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: a modular machine learning software library*. Idiap-RR Idiap-RR-46-2002. IDIAP.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS '08/ETAPS '08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[12] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. 2007. A Tutorial on Satisfiability modulo Theories. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV '07)*. Springer-Verlag, Berlin, Heidelberg, 20–36.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[14] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142. https://doi.org/10.1109/MSP.2012.2211477

[15] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3211346.3211349

[16] Bruno Dutertre. 2014. Yices 2.2. In *Computer-Aided Verification (CAV'2014) (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 737–744.

[17] Frederik Eaton. 2006. Statically Typed Linear Algebra in Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Portland, Oregon, USA) *(Haskell '06)*. Association for Computing Machinery, New York, NY, USA, 120–121. https://doi.org/10.1145/1159842.1159859

[18] Herbert B. Enderton. 2001. *A Mathematical Introduction to Logic, Second edition*. Elsevier.

[19] Matvey Ezhov. 2021. Simple dynamic seq2seq with TensorFlow. https://notebook.community/ematvey/tensorflow-seq2seq-tutorials/1-seq2seq.

[20] John Foderaro. 1991. LISP: Introduction. *Commun. ACM* 34, 9 (sep 1991), 27. https://doi.org/10.1145/114669.114670

[21] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

[22] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV '07)*. Springer-Verlag, Berlin, Heidelberg, 519–531.

[23] Yanjie Gao, Yonghao Zhu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2021. Resource-Guided Configuration Space Reduction for Deep Learning Models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (Madrid, Spain) *(ICSE '21)*. 175–187. https://doi.org/10.1109/ICSE43902.2021.00028

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[25] Google. 2008. Protocol Buffers - Google's data interchange format. https://developers.google.com/protocol-buffers/.

[26] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[27] Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3460944.3464310

[28] C. A. R. Hoare. 1971. Procedures and Parameters: An Axiomatic Approach. In *Symposium on Semantics of Algorithmic Languages*, E. Engeler (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–116.

[29] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[31] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[32] C. Barry Jay. 1995. A semantics for shape. *Science of Computer Programming* 25, 2 (1995), 251 – 283. https://doi.org/10.1016/0167-6423(95)00015-1

[33] C. Barry Jay and Milan Sekanina. 1997. Shape Checking of Array Programs. In *Proceedings of 1997 Computing: The Australasian Theory Symposium* (Sydney, Australia) *(CATS '97)*. Australian Computer Society, Inc., AUS.

[34] Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. *CoRR* abs/2010.07763 (2020). arXiv:2010.07763 https://arxiv.org/abs/2010.07763

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[36] Ibrahem Kandel and Mauro Castelli. 2020. The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. *ICT Express* 6, 4 (2020), 312–315. https://doi.org/10.1016/j.icte.2020.04.010

[37] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. 2018. Lightweight Verification of Array Indexing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/3213846.3213849

[38] Wen Kokke, Ekaterina Komendantskaya, Daniel Kienitz, Robert Atkey, and David Aspinall. 2020. Neural Networks, Secure by Construction. In *Proceedings of the 18th Asian Conference on Programming Languages and Systems (APLAS '20)*, Bruno C. d. S. Oliveira (Ed.). Springer International Publishing, Cham, 67–85.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS '12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[40] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[41] Rasmus Munk Larsen and Tatiana Shpeisman. 2019. TensorFlow Graph Optimizations.

[42] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https:

//www.usenix.org/conference/osdi21/presentation/lehmann
[43] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W. Senior, Fumin Wang, and Phil Blunsom. 2016. Latent Predictor Networks for Code Generation. *CoRR* abs/1603.06744 (2016). arXiv:1603.06744 http://arxiv.org/abs/1603.06744
[44] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the Interoperability between Deep Learning Frameworks by Model Conversion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1320–1330. https://doi.org/10.1145/3368089.3417051
[45] Yutaka Matsuno and Hiroyuki Sato. 2003. Flow Analytic Type System for Array Bound Checks. *Electronic Notes in Theoretical Computer Science* 78 (04 2003), 178–195. https://doi.org/10.1016/S1571-0661(04)81012-0
[46] Alexander Merritt. 2020. ONNX-Scala: Typeful, Functional Deep Learning / Dotty Meets an Open AI Standard (Open-Source Talk). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala* (Virtual, USA) *(SCALA 2020)*. Association for Computing Machinery, New York, NY, USA, 33.
[47] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (oct 1992), 40–51. https://doi.org/10.1109/2.161279
[48] Microsoft. 2021. ND-series Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/nd-series.
[49] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.
[50] Takayuki Muranushi and Richard A. Eisenberg. 2014. Experience Report: Type-Checking Polymorphic Units for Astrophysics Research in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 31–38. https://doi.org/10.1145/2633357.2633362
[51] David L. Olson and Dursun Delen. 2008. *Advanced Data Mining Techniques* (1st ed.). Springer Publishing Company, Incorporated.
[52] oneDNN. 2021. Understanding Memory Formats. https://oneapi-src.github.io/oneDNN/v2.5/dev_guide_understanding_memory_formats.html.
[53] ONNX. 2017. Open Neural Network Exchange. https://onnx.ai/.
[54] Stack Overflow. 2017. Negative dimension size caused by subtracting 3 from 1 for "Conv2D". https://stackoverflow.com/questions/41651628/negative-dimension-size-caused-by-subtracting-3-from-1-for-conv2d.
[55] Ran Pan. 2014. Tensor Transpose and Its Properties. *CoRR* abs/1411.1503 (2014). arXiv:1411.1503 http://arxiv.org/abs/1411.1503
[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, Vol. 32. Curran Associates, Inc., 8024–8035. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf
[57] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
[58] Leonardo Piñeyro, Alberto Pardo, and Marcos Viera. 2021. Structure verification of deep neural networks at compilation time. *Journal of Computer Languages* 67 (2021), 101074. https://doi.org/10.1016/j.cola.2021.101074
[59] PyTorch. 2019. Conv2d crashes with stride=0. https://github.com/pytorch/pytorch/issues/27598.
[60] PyTorch. 2020. The topological sorting algorithm for the graph transformation subsystem. https://github.com/pytorch/pytorch/blob/v1.5.1/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h#L26.
[61] PyTorch. 2020. The torch.nn.Conv2d API. https://pytorch.org/docs/1.5.1/nn.html#conv2d.
[62] PyTorch. 2021. Torchvision. https://github.com/pytorch/vision.
[63] PyTorch. 2021. Translation with a Sequence to Sequence Network and Attention. https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html.
[64] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, USA.
[65] Brennan Saeta, Denys Shabalin, Marc Rasi, Brad Larson, Xihui Wu, Parker Schuh, Michelle Casbon, Daniel Zheng, Saleem Abdulrasool, Aleksandr Efremov, Dave Abrahams, Chris Lattner, and Richard Wei. 2021. Swift for TensorFlow: A portable, flexible platform for deep learning. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 240–254.
[66] Sergio Guadarrama, Nathan Silberman. 2016. TensorFlow-Slim: A lightweight library for defining, training and evaluating complex models in TensorFlow. https://github.com/google-research/tf-slim.
[67] Olha Shkaravska, Marko van Eekelen, and Ron van Kesteren. 2009. Polynomial Size Analysis of First-Order Shapely Functions. *Logical Methods in Computer Science* Volume 5, Issue 2 (May 2009). https://doi.org/10.2168/LMCS-5(2:10)2009
[68] João P. Marques Silva and Karem A. Sakallah. 1997. GRASP—a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California, USA) *(ICCAD '96)*. IEEE Computer Society, USA, 220–227.
[69] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556
[70] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS '14)*. MIT Press, Cambridge, MA, USA, 3104–3112.
[71] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594
[72] TensorFlow. 2016. Large Strides for 1x1 Convolutions. https://github.com/tensorflow/tensorflow/issues/889.
[73] TensorFlow. 2018. Primitive Neural Net (NN) Operations. https://github.com/tensorflow/docs/tree/r1.8/site/en/api_docs/python/tf/nn.
[74] TensorFlow. 2019. The tf.layers.Conv2D API. https://github.com/tensorflow/docs/blob/r1.13/site/en/api_docs/python/tf/layers/Conv2D.md.
[75] TensorFlow. 2019. The tf.nn.convolution API. https://github.com/tensorflow/docs/blob/r1.13/site/en/api_docs/python/tf/nn/convolution.md.
[76] Takahisa Toda and Takehide Soh. 2016. Implementing Efficient All Solutions SAT Solvers. *ACM J. Exp. Algorithmics* 21, Article 1.12 (Nov. 2016), 44 pages. https://doi.org/10.1145/2975585
[77] Ben Trevett. 2021. PyTorch Seq2Seq. https://github.com/bentrevett/pytorch-seq2seq.
[78] Sahil Verma and Zhendong Su. 2020. ShapeFlow: Dynamic Shape Interpreter for TensorFlow. *CoRR* abs/2011.13452 (2020). arXiv:2011.13452 https://arxiv.org/abs/2011.13452
[79] WALA. 2021. The T. J. Watson Libraries for Analysis. https://github.com/wala/WALA.
[80] Wikipedia. 2021. Cross-correlation — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Cross-correlation&oldid=1031522391.
[81] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732
[82] Hongkun Yu, Chen Chen, Xianzhi Du, Yeqing Li, Abdullah Rashwan, Le Hou, Pengchong Jin, Fan Yang, Frederick Liu, Jaeyoun Kim, and Jing Li. 2020. TensorFlow Model Garden. https://github.com/tensorflow/models.
[83] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664
[84] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 783–794. https://doi.org/10.1109/ICSE.2019.00086
[85] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1159–1170. https://doi.org/10.1145/3377811.3380362
[86] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115. https://doi.org/10.1109/ISSRE.2019.00020
[87] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866