Checkpointing and Deterministic Training for Deep Learning

Xiangzhe Xu*
xzx@purdue.edu
Purdue University
USA

Hongyu Liu*†
liuhyscc@gmail.com
Huawei Galois Lab
Purdue University
China & USA

Guanhong Tao
taog@purdue.edu
Purdue University
USA

Zhou Xuan
xuan1@purdue.edu
Purdue University
USA

Xiangyu Zhang
xyzhang@purdue.edu
Purdue University
USA

ABSTRACT

Checkpointing and faithful replay are important for the training process of a Deep Learning (DL) model. It may improve productivity, model performance, robustness, and help security auditing. However, the inherent nondeterminism in training poses prominent challenges. Even with fixed random seeds, multiple runs of a same training pipeline may yield models whose performance varies by 20% percent. With existing infrastructural checkpointing support, developers cannot faithfully replay a training process. In this paper, we propose **DETrain**, a new solution to checkpointing and faithful execution/replay for long running DL training programs. We introduce a novel random number generation mechanism that can generate consistent random numbers in the presence of data parallelism. In addition, we devise a novel analysis that can determine a set of state variables that are necessary for faithful replay. These variables are either saved in a checkpoint or re-generated by fast forwarding, a selective execution technique. DETrain is evaluated on 13 PyTorch models and 16 Tensorflow models. It can deterministically execute these programs and replay from checkpoints with reasonable overhead. It also helps developers in diagnosing problems in training.

KEYWORDS

deep learning, determinism, record-and-replay system

1 INTRODUCTION

Deep Learning (DL) model training is considered an art, especially for large models and datasets. Setting up the appropriate hyper parameters, constructing the datasets, and choosing the right model architecture and capacity require substantial expertise. Even with all the appropriate settings, it is often the case that luck is also needed to acquire superb training results, because model accuracy is dependent on batch selection and training order, which are random in the current practice. In this process, a lot of heuristics are being utilized, for instance, constantly measuring model accuracy and saving a copy of the best performing one [11, 31]. In some situations, training data may even be mutated on the fly in order to achieve higher accuracy [8, 12]. In the end, even though a model with high accuracy is reported as the result, it is almost impossible for others to reproduce it, not to mention learning from the tuning procedure. In these cases, a checkpointing and replay technique can faithfully record the entire training and tuning procedure, helping transform the art of DL training to a science.

In addition, DL models are vulnerable to adversarial attacks [15, 37] and backdoor attacks [6, 14]. The former is often related to underfitting and/or overfitting during training. In backdoor attacks, any input stamped with a so-called *trigger* (e.g., a small patch with solid color) causes the model to misclassify the input to a specific target label. It is often associated with dirty data being intentionally/unintentionally used during training. While DL models are being deployed to a lot of applications in our daily life, including many that are safety-critical, such as self-driving vehicles and identity recognition, ensuring model security is a prominent challenge. An important aspect of model security is the capabilities of auditing a model's training procedure such that any adversary can be held accountable; and performing forensic analysis to understand the attack provenance. We argue in the future, when a pre-trained model is published, all the information that is needed for others to faithfully reproduce the model (e.g., checkpoints and random seeds) should be published as well.

Finally, DL model training is an extremely expensive procedure, consuming a substantial amount of resource in terms of CPU/GPU cycles, time, memory, and disk space. However, just like software has bugs, model implementation and even the underlying infrastructures inevitably have bugs. These bugs are often non-deterministic, meaning they may or may not manifest themselves in a particular execution. In addition, there are hardware outages and transient bit flips (e.g., caused by environmental condition changes). All of these may cause an interruption of the expensive training process. Checkpointing and deterministic replay are hence highly desirable functionalities for data engineers to protect their investment and fix their bugs.

Existing Checkpointing Support. Popular DL frameworks provide some basic checkpointing support. For example, in Tensorflow, the developers can create a callback function using a pre-defined

*Both authors contributed equally to this research.
†This work was initiated and conducted while Hongyu Liu was a postdoctoral associate at Purdue University

CAIN'22, May 16–24, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9275-4/22/05.
https://doi.org/10.1145/3522664.3528605

checkpointing API, which simply saves a copy of the model parameters. The callback function is then invoked at the end of every epoch, or after a fixed number of training steps. Upon exception, the developer can load the saved parameters and restart training from those parameters. Alternatively, the developers can write their own checkpointing function to save any information selected by themselves, which requires non-trivial human efforts and model-specific implementation. Neither approach supports faithful replay of the training procedure, from either the beginning or a checkpoint.

There is also traditional software checkpointing and deterministic replay support [7, 13, 16, 27], in which a checkpoint saves a comprehensive image of all active states of the program and additional runtime information (such as system input values from files and sockets) to facilitate deterministic replay. Such strong support enables a wide range of applications such as production system debugging [19, 24, 34], security attack forensics [26, 35], and fault tolerance [4]. One may argue that a DL training process can be considered a special software process, to which existing general software checkpointing and replay techniques can be applied. Unfortunately, taking a memory snapshot could easily consume tens/hundreds of gigabytes in modern DL training; and there are a lot of unique non-deterministic factors that traditional techniques cannot handle well.

Our Solution. Therefore, we develop a novel checkpointing and deterministic execution/replay technique for DL training. The technique possesses the traits from both the existing DL infrastructural support (e.g., concise checkpoints) and the traditional software checkpointing and replay techniques (e.g., automation and support of deterministic training). Given a model implementation, our technique leverages program analysis to automatically identify all the places that can induce non-determinism (e.g., random seed initialization and random number updates) and all the critical state information needed for faithful replay (e.g., shuffled dataset). These random number operations may be confounded by data parallelism such that using fixed random seeds cannot ensure deterministic execution. It then transparently patches/instruments the model implementation to add statements to save/restore such critical information. During training, the inserted statements regularly record runtime information. Upon exceptions or a forensics/audit request, the inserted statements restore the model training process to the same state as that when the checkpoint was created. For example, it can transparently recover the states of the various random number generators such that any following stochastic operations (e.g., random dropout) would behave as the original training process. For some states that are too expensive to record in checkpoints, e.g., shuffled datasets, our technique re-generates through a selective execution technique called *fast-forwarding*, in which the training process starts from the beginning and the training loop(s) repeat the same number of times as what is recorded in the checkpoint, performing only the computation for the states need fast-forwarding, *excluding expensive operations such as gradient computation and weight updates*. For example, to fast-forward data shuffling, the re-execution only needs to repeat the training epochs just for shuffling.

Our contributions are summarized as follows.

- We propose a novel checkpointing and faithful replay technique for DL training programs.
- We develop program analysis to identify sources of uncontrollable non-determinism and state variables that are needed for faithful replay. These analyses are described based on a simplified language modeling DL applications.
- We develop a runtime to support removing uncontrollable nondeterminism, saving checkpoints, fast-forwarding, and state restoration.
- We develop a prototype DETrain and evaluate it on 13 PyTorch and 16 Tensorflow model training real-world programs. Our results show that our tool can achieve its stated goals with reasonable overhead. We also perform three case studies to demonstrate how DETrain can provide substantial help to developers in problem diagnosis during training. DETrain can be found at our GitHub repository[1].

2 MOTIVATION

2.1 Background: Deep Learning Framework

Machine learning (ML) frameworks seamlessly connect high-level programming languages (e.g., Python) with the high-performance native code. To leverage an ML framework, developers specify the data-flow of the model. Then the framework adds code to compute the gradient, compiles the Python code to intermediate representations, delegates key operations to the native code, and dispatches operations to heterogeneous devices. We use Tensorflow as an example to illustrate the typical workflow of ML frameworks. As depicted by Fig. 1, Tensorflow first compiles the python code to a computation graph. Each node of this graph, known as an operator, is the atomic scheduling unit in Tensorflow. Then an executor dispatches nodes to devices where these nodes are executed concurrently.

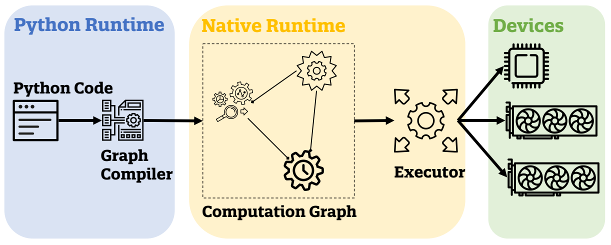


Figure 1: Workflow of Tensorflow

2.2 Motivation Example

To further illustrate how Tensorflow works and the challenges of deterministic replay, we introduce a running example. *Adversarial attack* is a technique to make a model mis-predict by perturbing inputs. The perturbed inputs are also called *adversarial examples*. To defend against adversarial attacks, developers usually harden a model by using adversarial examples in training. The process is called *adversarial training*. Fig. 2 shows a simplified Python code snippet for a well-known adversarial training technique, *Projected Gradient Descent* (PGD) [15]. It is slightly enhanced by co-training

[1]https://github.com/XZ-X/DETrain-public

the model with both benign and adversarial examples together [38] for better performance.

The training process iterates over the training dataset for 200 times. Each iteration is noted as an *epoch* (line 4). In each epoch, the dataset is shuffled and further split to small subsets, noted as *batches*. At line 6, the variables `batchX` and `batchY` are data and labels for a batch, respectively. For each batch, the algorithm first computes the cross-entropy loss, generates the gradients, and optimizes weights of the model by gradient descent. These operations are implicitly performed by the API `model.fit()` at line 7. Then it generates a set of adversarial examples in the function `adv_gen()`. After that, it further trains the model with the generated adversarial examples to make it robust. Now we take a close look at the two functions in the example. The `Model()` function at line 11 defines the model we want to train. It consists of four layers: an input layer that encodes input image to vectors, a deterministic layer that performs some deterministic computations, a dropout layer that randomly sets some values to zeros, and an output layer that computes the classification label. Note that the model is for illustration purpose. The real model is a ResNet50 wide[38] that is much more complex. When the function is invoked, the Tensorflow framework compiles these layers to a computation graph. The `adv_gen()` function at line 18 generates adversarial examples from the benign data `benignX`. It first adds some random noise to the benign data, then computes the gradient of the model's loss with respect to the input data. After that, it performs gradient ascent on the input to generate samples that can maximize the loss of the model, that is, trying to induce misclassification.

```
1   tf.global_seed = 123
2   data = get_dataset()
3   model = Model()
4   for epoch in range(0...200):
5       data = shuffle(data)
6       for batchX, batchY in data:
7           model.fit(batchX, batchY)
8           adv_examples = adv_gen(model, batchX, batchY)
9           model.fit(adv_examples, batchY)
10
11  def Model():
12      input = Input_layer()
13      tmp = Determ_layers(input)
14      tmp2 = Dropout(tmp, seed=234)
15      out = Output_layer(tmp2)
16      return tf.Model(input, out)
17
18  def adv_gen(model, benignX, benignY):
19      tmp = benignX + Random_noise(seed=567)
20      for _ in 0...30:
21          g = gradient(model(tmp, benignY), tmp)
22          # a sequence of deterministic operations
23          proj_g = project(g)
24          tmp += proj_g
25      return tmp
```

Figure 2: Motivation Example - PGD Adversarial Training

2.3 Sources of Nondeterminism

There are two major sources of nondeterminism: the *algorithmic ones* and the *non-algorithmic ones*. The former refers to the randomness that developers intentionally introduce to improve model generalization. For example, the dropout layer at line 14 randomly "drops" some values by setting them to zeros; the random noise generator at line 19 adds random noise to benign samples. The behaviors of these algorithmic nondeterminisms can be determined by setting random seeds.

On the other hand, non-algorithmic nondeterminism refers to the randomness that developers cannot control. The most significant non-algorithmic nondeterminism is caused by data parallelism. When a random generator is accessed by multiple threads concurrently, the random number obtained by any given thread is non-deterministic. For instance, at line 7, when `model.fit()` is called, Tensorflow feeds the whole batch of data to the computation graph (compiled from the model code). Then the Tensorflow native code dispatches these computation tasks to multiple physical devices. Note that the random generator for the dropout layer is shared by all threads. That means, in different executions, a given instance of data may undergo different dropout behaviors, which further affect the result of the whole training. In our experiment, after fixing the global seed, two executions of the running example yield two models whose robust accuracy (using the C&W attack [1]) may differ by 20% in the worst case (28% versus 48%), while having comparable benign accuracy (87%). Our technique allows deterministically replaying these executions to study the underlying reasons for the inconsistent performance. As we will show in Section 6.4, it can be used to replay non-deterministic bugs and facilitate repair.

3 DETERMINISTIC EXECUTION

In the previous section, we mention that model training programs are haunted by non-algorithmic nondeterminism even with fixed random seeds. In this section, we use a language to model training programs and then describe our analysis and instrumentation to ensure determinism using the language.

3.1 Language

To facilitate discussion, we introduce a language to represent model training programs. For simplicity, we only model elements related to nondeterminism. The syntax of our language is depicted in Fig. 3.

In our language, the top-level components of a program are statements. Each statement models a core operation used in model training and is associated with a label L. We introduce statements to describe the building, inferring, and training process related to models: $M =^L$ **build**(S) builds a model M from the descriptions in statement S; $V_1 =^L M(V_2, V_3)$ computes the loss of model M on data V_2 and label V_3, then the loss is stored in V_1; $B^L(M, V)$ represents backward propagation of gradient V on model M. To model random number generation, we introduce two statements: $G =^L$ **create_generator**(V) creates a random number generator G with an initial seed V; $V =^L$ **random_gen**(G, E) generates random numbers with the same shape as E using generator G. Moreover, we use **data_parallel_begin**$^L(V)$ and **data_parallel_end**^{L} to denote the begin and end of a data parallel region. In a data parallel region, ML frameworks divide data V into subsets and perform operations on these subsets concurrently. Finally, **checkpoint**$^L(V_1, V_2, ...)$ means saving the value of variables $V_1, V_2, ...$ in the checkpoint.

Besides statements, we also introduce two special expressions to represent domain specific behaviors in model training. $\nabla(E, V)$ denotes the gradient of E w.r.t. the variable V; $V[E]$ denotes a subset of data in V or a permutation of data in V. For example, suppose

$$
\begin{array}{llcl}
\langle\texttt{Random Generator}\rangle & G & ::= & \{rng1,\ rng2,\ \ldots\} \\
\langle\texttt{Dataset}\rangle & D & ::= & \{ds1,\ ds2,\ \ldots\} \\
\langle\texttt{Model}\rangle & M & ::= & \{model1,\ model2,\ \ldots\} \\
\langle\texttt{Variable}\rangle & V & ::= & \{x,\ y,\ \ldots\} \\
\langle\texttt{Const}\rangle & C & ::= & \{0,1,2,\ldots\} \\
\langle\texttt{Label}\rangle & L & ::= & \{L_1,\ L_2,\ \ldots\} \\
\langle\texttt{Program}\rangle & P & ::= & S \\
\langle\texttt{Statement}\rangle & S & ::= & S_1;S_2 \mid V =^L E \mid V[E_1] =^L E_2 \mid B^L(M,V) \\
 & & \mid & \texttt{for}\ (V_1{=}E_1\ \texttt{to}\ E_2)^L\ \{\ S\ \} \\
 & & \mid & G =^L \texttt{create\_generator}(V) \\
 & & \mid & V =^L \texttt{random\_gen}(G,\ E) \\
 & & \mid & \texttt{data\_parallel\_begin}^L(V) \\
 & & \mid & \texttt{data\_parallel\_end}^L \\
 & & \mid & V_1 =^L \texttt{generate\_batches}(V_2) \\
 & & \mid & M =^L \texttt{build}(S) \\
 & & \mid & V_1 =^L M(V_2,V_3) \\
 & & \mid & D =^L \texttt{load\_data}() \\
 & & \mid & \texttt{checkpoint}^L(V_1,V_2,\ldots) \\
\langle\texttt{Expression}\rangle & E & ::= & C \mid V \mid V[E] \mid (E_1,\ E_2) \mid E_1\ OP\ E_2 \mid \nabla(E,V) \\
\langle\texttt{Operator}\rangle & OP & ::= & + \mid - \mid * \mid / \mid \ldots
\end{array}
$$

Figure 3: Syntax of our Language

that $V = [a,b,c]$. Then $V[0]$, $V[1,2]$ can represent two subsets of V, which are $\{a\}$ and $\{b,c\}$, respectively; and $V[2,0,1]$ represents a permutation of V, which is $\{c,a,b\}$.

The other parts of our language have similar meanings with their counterparts in a commonplace programming language. Thus we omit their discussion for simplicity.

3.2 Execution Model of ML Frameworks

In this section, we revisit our motivation example to illustrate the execution model of ML frameworks. Specifically, we demonstrate how the frameworks compiles operations to graphs and how they empower the training process with data parallelism. Finally, we discuss how non-algorithmic nondeterminism is introduced.

Fig. 4 presents the motivation example rewritten in our language. It first creates a global random number generator at line 2 with the seed specified by developers. After that, it generates a local random generator for each random operation. For example, at line 4, rng1 is created for the dropout operation. Note that in the case that developers do not specify a seed for a random operation, a seed will be generated from the global random number generator, as shown at line 7. At line 12, the build() function stands for the framework compiling the model to a computation graph. The data fed to the model will undergo the operations specified in the statements inside build(). Line 19 starts the main training loop where the model iterates over the dataset for 200 times. In each iteration, a random sequence is generated to shuffle the dataset. Then the dataset is further divided into batches, and the loop starting at line 23 iterates over the batches.

For each batch, the framework leverages data parallelism to accelerate the training process, as depicted by Fig. 5. The green rectangle on the left represents the data in a batch, and the blue rectangles denote the operations that will be performed on each data example. The framework builds a graph to represent the required computations, denoted by the grey rectangle. Suppose we have two threads. The framework dispatches an execution task to each of them, with each task containing a subset of data, a thread local execution state, and a reference to the computation graph. During execution, both threads fetch data from their data subsets, and concurrently execute the graph on these data. Since the graph

```
 1  # the global random generator
 2  rng0 = create_generator(123)
 3  # local generator for dropout
 4  rng1 = create_generator(234)
 5  # local generator for shuffle
 6  shuffle_seed = random_gen(rng0, 1)
 7  rng2 = create_generator(shuffle_seed)
 8  # local generator for random noise
 9  rng3 = create_generator(567)
10
11  ds1 = load_data()
12  model1 = build(
13      ... # other operations on input
14      drop_mask = random_gen(rng1, size(input)/2);
15      out = input[drop_mask];
16      ... # other operations on out
17  )
18  data = ds1
19  for epoch = 0 to 200{
20      shuffle_mask = random_gen(rng2, size(data))
21      data = data[shuffle_mask]
22      batch = generate_batches(data)
23      for i = 0 to size(batch){
24          current_batch = batch[i]
25          data_parallel_begin(current_batch)
26              X = current_batch[0]
27              Y = current_batch[1]
28              loss = model1(X, Y)
29              g = ∇(loss, model1) # compute the gradient
30              B(model1, g) # backward propagation
31              noise = random_gen(rng3, size(current_batch))

32              adv_sample = X + noise
33              for k = 0 to 30{
34                  loss = model1(adv_sample, Y)
35                  g = ∇(loss, adv_sample)
36                  ... # other operations on g
37                  adv_sample = adv_sample + g
38              }
39              # adversarial training
40              loss = model1(adv_sample, Y)
41              g = ∇(loss, model1)
42              B(model1, g)
43          data_parallel_end
44          # suppose developers want to save model1
45          checkpoint(model1)
46      }
47  }
```

Figure 4: Motivation Example in Our Language

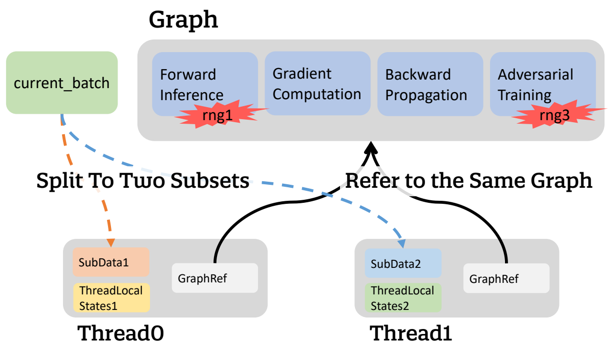


Figure 5: Data Parallel

is shared by both threads, the generators rng1 and rng3 are also shared and concurrently invoked (in line 14 and line 31, respectively) Hence for either thread, the random numbers it acquires depend on the order in which the threads access these generators. In other words, the random numbers are nondeterministic since the access order is not fixed among executions despite of the fixed global seed.

3.3 Elimination of Nondeterminism

We develop a program analysis to determine the places that are susceptible to nondeterminism and instrument them to eliminate such nondeterminism. From the above discussion, we know that the undesirable nondeterminism is rooted at the use of random number generators that are created outside of a data parallel code region for data parallel computation. We hence develop a program analysis to determine if the use of a random number generator is thread-safe. Thread-unsafe uses of genenators will hence be replaced by thread-safe uses. Specifically, we observe that the finest granularity of data parallelism is at the example level, that is, the processing of any pair of examples could be parallel, whereas the operations on a particular example have to be sequential. We hence pre-generate a random number generator for each data example and replace thread-unsafe uses with queries to the generators associated with individual examples. In the remainder of the section, we focus on the analysis.

$\textcircled{a}, \textcircled{b}$ ∈ `Enum`

TS ∈ `ThreadSafe` ::= `Generator` × ($\textcircled{a}$ + $\textcircled{b}$) × `Label` → `Boolean`

ERR ∈ `ErraticRandomGenerator` ::= `Generator` → `Boolean`

`succ`(L): This function returns labels of the successors of the statement whose label is L.

`pred`(L): This function returns labels of the predecessors of the statement whose label is L.

`begin`(L): This function is defined only when L is the label of a **`data_parallel_end`** statement. It returns labels of the related **`data_parallel_begin`** statement.

Figure 6: Abstract Domain for Elimination of Nondeterminism

We describe our analysis as solving a set of constraints, which yield results in an abstract domain that denotes if a generator is thread-safe.

Definitions. The abstract domains are shown in Fig. 6. Here, symbols $\textcircled{b}$ and $\textcircled{a}$ refer to the program points before and after a label, respectively. TS indicates whether a random generator is safe to use concurrently at a given program point. ERR stores the random generators that are not thread-safe (and hence require fixing). We also define a set of auxiliary functions to simplify our analysis description. Functions **`succ`**(L) and **`pred`**(L) refer to two sets of labels denoting the successors and the predecessors of statement L, respectively; **`begin`**(L) is defined only when L is the label of a **`data_parallel_end`** statement. It refers to the paired **`data_parallel_begin`** statement.

Analysis Rules. The analysis rules are shown in Table 1. Intuitively, our rules detect problematic accesses (to generators) by introducing a thread local scope for the data parallel region. **Random_Create** ensures the created generator G is safe to use (within the current scope) after the creation statement. **Parallel_Region** consists of two rules: upon entering the data parallel region, all generators defined outside become unsafe to use since they may yield racy results; upon exit, these generators become safe again. **Random_Use** guarantees all generators with unsafe usage being marked in ERR. **Default** propagates the (flow-sensitive) thread safety information through other statements. A generator is safe to use *before* the current statement only if it is safe *after* all the predecessors.

Example. Consider the example in Table 2. The line numbers (corresponding to the code in Fig. 4), the related statements, and a solution to the constraints are shown in the columns from left to right. Note that we omit irrelevant values in the abstract domain for simplicity. At line 9, generator $rng3$ is created, and thus it is safe to use after this program point. Then the constraints for the statement at line 25 guarantee that $rng3$ is invalidated after line 25 since it is created outside the parallel region. When the code in the parallel region tries to use $rng3$ at line 31, the precondition $TS[rng3\textcircled{b}31] = False$ is satisfied. Hence $ERR[rng3]$ should be $True$ in the solution. In this way, the solution successfully captures the unsafe use of $rng3$ and marks it in ERR. Similarly, at line 28, the model is invoked to compute the loss, which further executes the random generation statement at line 14. Since $rng1$ is also unsafe to use in the parallel region, $ERR[rng1]$ is $True$ as well.

After capturing all the unsafe generators, we replace them with pre-generated per-example random number generators. The loop index of epoch and the id of example are used to deterministically look up a unique seed for this local generator. With the deterministic seed, all the random number queries for computation related to the data sample become deterministic, leading to a fully deterministic execution. Details are elided.

4 CHECKPOINTING AND REPLAY

In this section, we introduce a novel technique that can create efficient checkpoints and support faithful replay. Our technique assumes that the developers already make use of built-in "checkpointing" APIs to save copies of certain states (e.g., weight values). It leverages the information conveyed by such API invocations, such as the intended places for checkpointing and the program states that are already recorded (by the built-in APIs), and then analyzes and determines the other information needed to enable faithful replay. Specifically, it automatically saves random number generator states (e.g., how many random numbers have been generated from a particular seed) and the number of executed iterations of training loops. For other heavy-weight but needed states, such as a shuffled dataset, it *re-generates* such states by *fast-forwarding*. For example, we regenerate the shuffled states of a dataset by rerunning the same number of epochs and only executing the shuffling operation, *without executing any other expensive training operations*. The key insight is that the results of the expensive training operations can be restored from the checkpoint. Hence, the challenge lies in identifying the states that need fast-forwarding. Note that if we decide to fast-forward a variable x, we need to fast-forward any of the variables that x depends on. We leverage an analysis to identify all the variables that need fast-forwarding and our system automatically re-generates these states (by executing from the beginning). We call it fast-forwarding as these states only require very lightweight computation to regenerate.

We develop a program analysis to determine the variables that need to be fast forwarded. In a "checkpointing" statement inserted by the developer, such as `tf.train.Saver.save()`, only some states are saved, such as weight values. If there are other states computed before the "checkpoint" but used in computation beyond the

Table 1: Analysis Rules for Nondeterminism

Rule	Statement	Constraints
PARALLEL_REGION	**data_parallel_begin**$^L(V)$	$\forall g, TS[g\textcircled{b}L] = \bigcap_{i \in pred(L)} TS[g@i] \land \forall g, TS[g@L] = False$
	data_parallel_end^L	$\forall g, TS[g@L] = TS[g\textcircled{b}begin(L)]$
RANDOM_CREATE	$G =^L$ **create_generator**(V)	$\forall g, TS[g\textcircled{b}L] = \bigcap_{i \in pred(L)} TS[g@i]\ TS[G@L] = True \land \forall g \neq G, TS[g\textcircled{b}L] = TS[g@L]$
RANDOM_USE	$V =^L$ **random_gen**(G, E)	$\forall g, TS[g\textcircled{b}L] = \bigcap_{i \in pred(L)} TS[g@i] \land TS[G\textcircled{b}L] = False \rightarrow ERR[G] = True$
DEFAULT	Other Statements	$\forall g, TS[g\textcircled{b}L] = \bigcap_{i \in pred(L)} TS[g@L] \land \forall g, TS[g\textcircled{b}L] = TS[g@L]$

Table 2: Example for Elimination of Nondeterminism

LineNo.	Statement	Abstract Domain	
		TS	ERR
9	`rng3 = create_generator(567)`	$TS[rng3@9] = True$	
…	…	…	…
25	`data_parallel_begin(current_batch)`	$TS[\{rng1, rng3\}\textcircled{b}25] = True; TS[\{rng1, rng3\}@25] = False$	
14	`drop_mask = random_gen(rng1, size(input)/2)`	$TS[rng1\textcircled{b}14] = False$	$ERR[\{rng1, rng3\}] = True$
…	…	…	
31	`random_gen(rng3, size(current_batch))`	$TS[rng3\textcircled{b}31] = False$	$ERR[\{rng1, rng3\}] = True$
…	…	…	…

"checkpoint", we will need to restore them in order to achieve faithful replay. Our analysis identifies such states (for fast-forwarding). We describe our analysis as solving a set of constraints, which yield the variables that are needed by faithful replay but are not included in the developer "checkpoint". The statements related to defining these variables will be re-executed during fast forwarding.

$@, \textcircled{b} \in$ `Enum`
$Needed \in$ `NeededByFollowingCode ::=`
`(Variable+Model+RandomGenerator+Dataset)` $\times$ $(@ + \textcircled{b})$ $\times$ `Label`
$\longrightarrow$ `Boolean`

`Vars`$(expr)$: This function returns variables used in $expr$.

`succ`(L): This function returns labels of the successors of the statement whose label is L.

`pred`(L): This function returns labels of the predecessors of the statement whose label is L.

Figure 7: Abstract Domain for Checkpointing

Definitions. The definitions for the abstract domain are shown in Fig. 7. Besides the definitions in Fig. 6, we further introduce $Needed$ to indicate the variables need to be recomputed and an auxiliary function $\texttt{Vars}(expr)$ to refer to the set of variables used in $expr$.

Analysis Rules. The analysis rules are listed in Table 3. Intuitively, our rules guarantee the state of a variable is either restored from a checkpoint or recovered by fast forwarding. The third rule in **CHECKPOINTING** excludes the saved variables (by the developer checkpoint) from recomputing since they can be restored from the checkpoint, by setting the $Needed$ values of all the variables stored in the checkpoint to $False$ before the checkpoint statement. Note that this enables marking all other variables that are exclusively and transitively used in computing these variables not needed. The first set of rules in **STATEMENTS** describes the constraints for an assignment statement $V =^L E$. Specifically, the first rule dictates that if V is not used in E, V is not needed before L. Intuitively, it indicates that any *previous* definition of V is not needed right before L as V is about to be re-defined at L. The second rule specifies that if V is not needed after L, the necessities of all variables after L and before L are identical. The third rule asserts that if V is needed after L, then all variables in E are needed before L; for variables other than V and those in E, their necessities before and after L are equivalent. The rules for $V[E_1] =^L E_2$ are similarly defined. For the gradient back-propagation statement $B^L(M, V)$, its semantics is similar to $M = ...(M, V)$. Hence, its rules are defined accordingly. The rules for other statements are similar and hence elided. The **FLOW** rule depicts how the necessity information is propagated/merged along control flow. In particular, if a variable is needed *before any* successor of the current statement, this variable is needed *after* the current statement. This rule applies to all statements. At the end, all the statements that compute some needed state, that is, whose left-hand-side variable has a $True$ value in $Needed$ right after these statements need to be re-executed. Our analysis shares some common nature with liveness analysis in compilers [17]. The difference lies in that our analysis does not simply consider any right-hand-side variable needed. Instead, the necessity is derived from the left-hand-side variable's necessity.

Example. Consider the example in Table 4. The line numbers (corresponding to the code in Fig. 4), the related statements, and a solution to the constraints are shown in the columns from left to right. At line 45, `model1` is saved to a checkpoint. Thus even if `model1` is needed after the checkpoint, we can restore its state from the saved file and hence its necessity before line 45 is set to $False$. In this way, we avoid repeating the expensive computation needed to train the model. As shown in the example, since `model1` is not needed *before* line 45, the operations at line 28, line 29, and line 30 are not needed and hence skipped in fast forwarding. On the other hand, the necessity of `data` is $True$ before line 21. Such necessity is propagated along the loop path line 45$\longrightarrow$ 19 $\longrightarrow$ 21 such that `data` is $True$ after the checkpoint line 45. Since `data` is not recorded in the checkpoint, it has to be fast-forwarded.

5 IMPLEMENTATION

Training programs heavily utilize DL frameworks through their Python interfaces. The behaviors that we are interested in, such as random number generation, are largely encapsulated inside framework APIs. It is not realistic to model the large number of APIs (for analysis). Hence, our system analyzes the training program and the framework code together. This entails two challenges. First, statically analyzing Python code is difficult due to its dynamic nature. In addition, a large part of these frameworks is implemented in

Table 3: Analysis Rules for Checkpointing

Rule	Statement	Actions
CheckPointing	$\textbf{checkpoint}^L(V_1, V_2, ...)$	$checked := \{V_1, V_2, ...\}$ $\forall v \notin checked, Needed[v\textcircled{b}L] = Needed[v@L]$ $\forall v \in checked, Needed[v\textcircled{b}L] = False$
Statements	$V =^L E$	$V \notin vars(E) \rightarrow Needed[V\textcircled{b}L] = False$ $Needed[V@L] = False \rightarrow \forall v, Needed[v\textcircled{b}L] = Needed[v@L]$ $Needed[V@L] = True \rightarrow \forall v \notin \{V\} \cup vars(E), Needed[v\textcircled{b}L] = Needed[v@L] \wedge \forall v \in vars(E), Needed[v\textcircled{b}L] = True$
	$V[E_1] =^L E_2$	$used := vars(E_1) \cup vars(E_2)$ $V \notin used \rightarrow Needed[V\textcircled{b}L] = False$ $Needed[V@L] = False \rightarrow \forall v, Needed[v\textcircled{b}L] = Needed[v@L]$ $Needed[V@L] = True \rightarrow \forall v \notin used \cup \{V\}, Needed[v\textcircled{b}L] = Needed[v@L] \wedge \forall v \in used, Needed[v\textcircled{b}L] = True$
	$B^L(M, V)$	$Needed[M@L] = False \rightarrow \forall v, Needed[v\textcircled{b}L] = Needed[v@L]$ $Needed[M@L] = True \rightarrow \forall v \notin \{V, M\}, Needed[v\textcircled{b}L] = Needed[v@L] \wedge \forall v \in \{V, M\}, Needed[v\textcircled{b}L] = True$
Flow	*	$\forall v, Needed[v@L] = \bigcup_{s \in succ(L)} Needed[v\textcircled{b}s]$

Table 4: Example for Checkpointing

LineNo.	Statement	Abstract Domain
		Needed
21	`data = data[shuffle_mask]`	$Needed[data\{\textcircled{b}, @\}21] = \mathbf{T}; Needed[\{X, Y, loss, g, model1\}\{\textcircled{b}, @\}21] = \mathbf{F}$
...	...	...
28	`loss = model1(X, Y)`	$Needed[data\{\textcircled{b}, @\}28] = \mathbf{T}; Needed[\{X, Y, loss, g, model1\}\{\textcircled{b}, @\}28] = \mathbf{F}$
29	`g = ∇(loss, model1)`	...
30	`B(model1, g)`	...
...	...	...
45	**`checkpoint`**`(model1)`	$Needed[data\textcircled{b}45] = \mathbf{T}; Needed[\{X, Y, loss, g, model1\}\textcircled{b}45] = \mathbf{F}; Needed[\{data, model1\}@45] = \mathbf{T}; Needed[\{X, Y, loss, g\}@45] = \mathbf{F}$
...	...	...

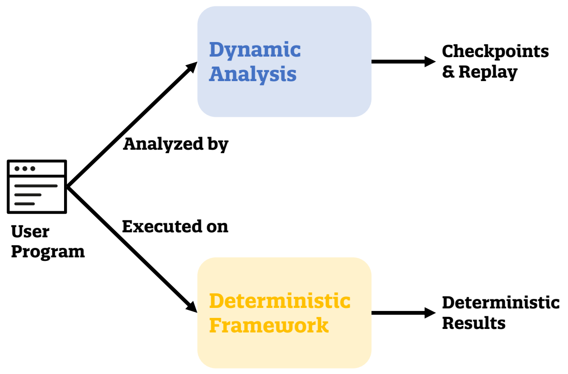


Figure 8: Workflow of DETrain

C/C++. As such, our analysis has to deal with (complex) Python and C/C++ code together. To achieve this goal, we resort to dynamic analysis. The workflow of DETrain is shown in Fig. 8. Leveraging the analysis described in Section 3, we empower ML frameworks to support deterministic execution. Our deterministic runtime supports thread-safe random number generator creation and replaces all thread-unsafe generators with the safe ones. Then we instrument the execution of training programs to support checkpoints and faithful replay. Specifically, we trace a number of key operations for model training, e.g., dataloading, gradient computation, and backward propagation, by intercepting API invocations. One can consider that the traces are essentially statement instances of the language in Section 3.1. Our analysis is hence performed on the traces. Note that DL training programs are different from general purpose software. It is easy to achieve full coverage of the training pipeline with a small input. Hence, our system does not suffer from the coverage problem. In addition to the tracer and the analysis, our checkpointing runtime saves random number generator states, epochs and steps. During fast-forwarding, the replay runtime re-executes the training loops with *almost empty loop bodies*, except those statements that need fast forwarding. After fast forwarding, the developer's checkpoint is loaded and the execution is resumed (for faithful replay).

6 EVALUATION

In this section, we evaluate DETrain and answer the following research questions.

- **RQ1:** Whether DETrain can deterministically execute and replay training?
- **RQ2:** To what extent DETrain affects the performance of training?
- **RQ3:** Is DETrain useful in real-world scenario?

To answer **RQ1** and **RQ2**, we train 29 real-world DL models with various model structures and datasets using DETrain. Three case studies are performed to illustrate the usefulness of DETrain, and thus answer the **RQ3**.

6.1 Experiment Setup

Environments. We perform the experiments on a server with two Intel Xeon Gold 6138 processors and an NVIDIA Tesla P100 16GB GPU. The OS is Ubuntu 18.04, installed with Linux-4.15.0.

Benchmarks. We evaluate DETrain on 29 popular models, such as BERT, ResNet50, VGG19, and so on. These models are collected from PyTorch examples [23], Tensorflow models [30], and popular repositories on Github [9, 36]. Among them, 13 models are built in PyTorch, and 16 models are built in Tensorflow. They are trained with widely-used datasets, such as ImageNet [2], SQuAD [25], and MovieLens [5]. *All the training programs are non-deterministic, meaning that they produce different models in multiple training runs.*

Note that the unsafe use of random number generator is not the only source of non-algorithmic nondeterminism [20]. For example, the order of floating point operations and the implementation for certain algorithms may also introduce nondeterminism. ML frameworks have good support to eliminate these nondeterminism. Thus in the experiments, we simply set some flags to benefit from the deterministic execution strategies provided by these frameworks.

6.2 RQ1: Deterministic Execution and Replay

To validate DETrain supports deterministic execution and replay, we conduct two experiments on each of the 29 model training programs. First, we run each training program for two times from the beginning and compare whether the two resulting models are exactly the same. Second, for each model, we arbitrarily select 4 checkpoints to recover from and continue the training process to the same epoch. Then the weights of the models are compared among different executions to ensure the determinism of the replays. For each of the 29 models, all the aforementioned 2+4 training runs yield the same weight values. It verifies that DETrain supports not only deterministic execution from the beginning, but also faithful replay from checkpoints.

6.3 RQ2: Performance of DETrain

Space Efficiency. Table 5 shows the characteristics of the subject models. The models, numbers of weight parameters, datasets used to train the models, and size of the vanilla checkpoints supported by the frameworks are shown in the columns from left to right. The last column presents the sizes of checkpoints created by DETrain. Overall, our checkpoints are about 10 kilobytes larger than the vanilla checkpoints for Pytorch models. The storage overhead of our checkpoint is thus negligible. On the other hand, our checkpoints for Tensorflow models incur more overhead. They consume more than one gigabyte space for six models and 100 megabytes for four models. The largest checkpoint is around two gigabytes for the RetinaFace ResNet50 model, while the largest overhead is for the DenseNet model, which is 1 gigabyte (Our Checkpoint)-1.5 megabytes (Checkpoint). We find two main reasons for the relatively large storage overhead. (1) We store all the states for random number generators since they typically have a small number of state values. However, in some models, there are a large number of generators, leading to significant space consumption. (2) Sometimes, the dataset is partitioned to multiple small buffers. Our analysis recognizes these buffers as plain variables, leading to unnecessary space consumption. The second problem can be overcome by better recognizing input examples (e.g., with developers' help). The first problem is caused by Tensorflow creating an independent random number generator for each random operation. For models with many layers, a significant number of generators are created. Note that the thread local generators we introduced are discarded upon exiting the data parallel region. They hence do not contribute to the overhead.

Time Efficiency. To evaluate the time efficiency of DETrain, all models are trained for more than an hour. Checkpoints are saved for roughly every 15 minutes. We compare our tool with executions on the vanilla PyTorch and Tensorflow. Figure 9a shows the performance overhead in PyTorch. Overall, the average overhead is around 15%. We observe that the overhead for most models is less than 20%. While for AlexNet and VGG, our tool imposes nearly two times runtime overhead. We further investigate the reasons for these two models. The slowdown primarily results from the deterministic cuDNN flag, which utilizes the deterministic implementations in Pytorch. Interestingly, we notice that ShuffleNetV2 and SqueezeNet achieve better performance, e.g., abound 5% faster. The results align well with the debate that programs with the deterministic cuDNN flag set may be more efficient than that with the default value, depending on whether cuDNN can always select the optimal implementation based on heuristics [21]. Figure 9b displays the overhead in Tensorflow. Our tool introduces 46% overhead on average. For some models, our tool incurs modest overhead. For BERT, BiT, EDSR, Sentiment Analysis model, and XLNET, it imposes less than 10% overhead. We believe that these models do not contain a lot of non-deterministic sources. Our tool runs slower for a few models. For ResNet-based models, the overhead is nearly two times. Further inspection shows that our tool instruments some non-deterministic algorithms, and forces Tensorflow to use deterministic ones. For example, the loss function `sparse_softmax_cross_entropy_with_logits` is not deterministic, our tool replaces this function with a deterministic implementation suggested in [29], which imposes the overhead. In general, the time overhead is rooted from: (1) uses of deterministic algorithms that are usually slower than their nondeterministic counterparts; (2) saving checkpoints. Note that (1) is not a limitation of our tool, but rather due to the change of nature in computation.

6.4 RQ3: Case Studies

We use three case studies to illustrate the advantages of our tool.

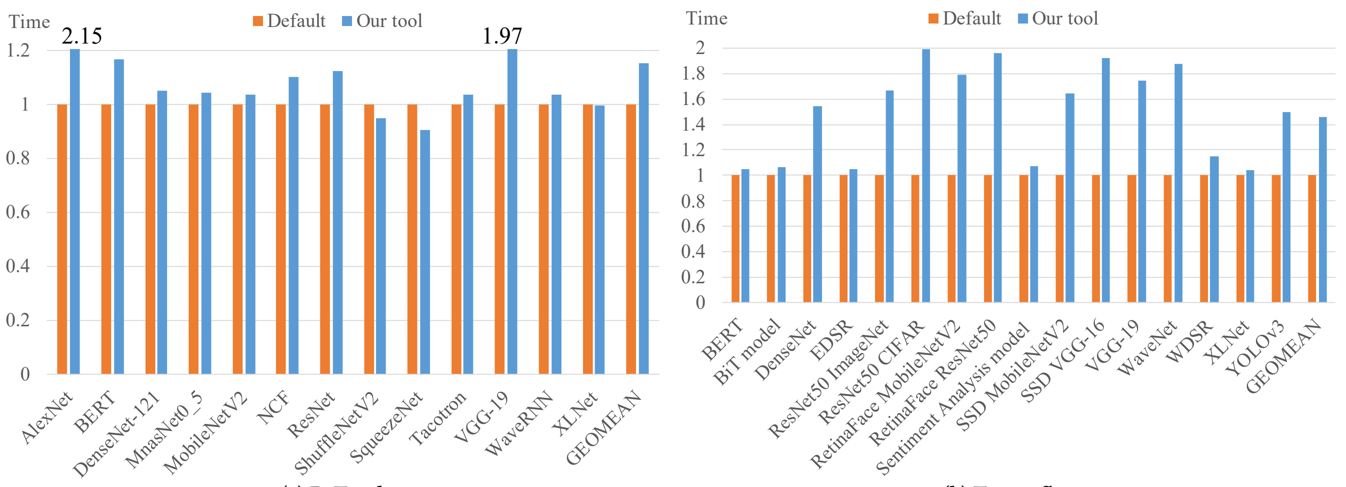
Debugging An Exception. We use a program that trains a ResNet18 model [32] on the CIFAR-10 dataset as a case study to demonstrate the advantages of debugging an exception with our tool. The exception causes the evaluation accuracy to suddenly drop after one hour of training. However, the exception cannot be deterministically reproduced, severely obstructing the debugging process. To illustrate the challenge to reproducing the bug, we show two training processes in Figure 10a, noted as `train1` and `train2`. Note that these two training instances start from exactly the same initial parameters and the same random seeds. The horizontal axis is the training epoch and the vertical axis is the accuracy. In `train1`, the bug does not appear at all, while it happens after 62500 iterations in `train2`. On the other hand, with the support of DETrain, we can stably reproduce the bug at the 54500th iteration, noted as `train-det` in Figure 10a.

After locating the bug, we further add some debugging code to examine gradient changes around the problematic iterations, shown in Figure 10b. We use the L1 norm to display the changes. Note that the gradients become very large at the 54500th iteration. Large gradients lead to aggressive weight updates, causing dramatic fluctuation in accuracy. To fix the bug, the learning rate should be adjusted, e.g., by utilizing an adaptive learning rate. After we set a smaller learning rate before the aggressive update happens, the bug disappears.

Adversarial Training. In the second case study, we run a randomized version of FGSM adversarial training [33] on MNIST to

Table 5: Characteristics of models

	Models	Parameters	Dataset	Checkpoint(KB)	Our Checkpoint(KB)
PyTorch	AlexNet	61,100,840	ImageNet	477,356	477,367
	BERT	108,311,810	SQuAD 2.0	1,264,770	1,264,780
	DenseNet-121	8,062,625	ImageNet	62,869	62,880
	MnasNet0_5	2,241,676	ImageNet	17,510	17,522
	MobileNetV2	3,539,036	ImageNet	27,595	27,607
	NCF	392,601	MovieLens 1M	4,606	4,617
	ResNet	25,610,205	ImageNet	199,950	199,961
	ShuffleNetV2	1,374,800	ImageNet	10,794	10,804
	SqueezeNet	1,248,424	ImageNet	9,772	9,783
	Tacotron	11,095,839	LJSpeech-1.1	130,065	130,076
	VGG-19	143,667,240	ImageNet	1,122,413	1,122,424
	WaveRNN	4,239,073	LJSpeech-1.1	49,685	49,697
	XLNet	119,083,010	SQuAD 2.0	1,386,375	1,386,382
TensorFlow	BERT	108,311,810	SQuAD 2.0	1,264,840	1,270,772
	BiT model	8,028,746	CIFAR10	237,578	1,502,874
	DenseNet	181,210	CIFAR10	1,481	1,228,255
	EDSR	1,517,571	DIV2K	17,830	32,930
	ResNet50 ImageNet	25,610,152	ImageNet	199,968	218,794
	ResNet50 CIFAR	856,058	CIFAR10	6,811	270,879
	RetinaFace MobileNetV2	2,223,872	WIDER FACE	14,138	1,967,339
	RetinaFace ResNet50	27,320,160	WIDER FACE	213,842	2,165,502
	Sentiment Analysis model	4,389,890	IMDB	34,293	50,839
	SSD MobileNetV2	8,493,678	VOC2012	99,818	473,583
	SSD VGG-16	26,285,486	VOC2012	308,078	518,493
	VGG-19	143,667,240	ImageNet	1,683,625	2,012,616
	WaveNet	3,128,875	LibriSpeech	36,657	36,693
	WDSR	615,088	DIV2K	7,424	22,524
	XLNet	119,083,010	SQuAD 2.0	1,395,641	1,407,137
	YOLOv3	61,949,149	VOC2012	726,366	1,782,031

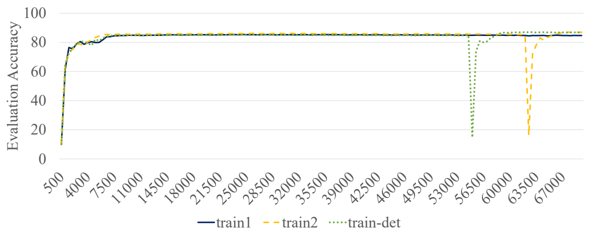


(a) PyTorch **(b) Tensorflow**

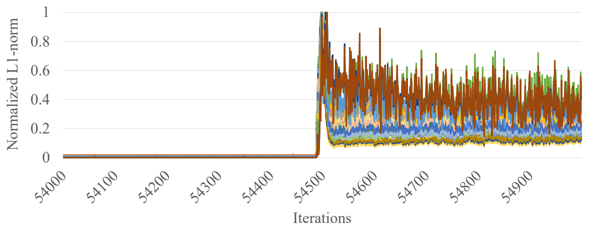
Figure 9: Runtime overhead.

illustrate the advantages of our tool. The FGSM algorithm initializes a perturbation using a hypercube with a radius $\epsilon/2$, and the step size is $\epsilon/2$, where ϵ is 0.3. The model consists of two convolutional networks with 16 and 32 output channels followed by a fully-connected layer with an output of size 100. We evaluate the robustness of the model using the PGD attack with $\epsilon = 0.3$.

Figure 11a shows the results when we train the model ten times without our tool. The evaluation robust accuracy is between 0.03%

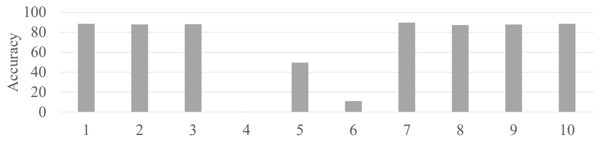


(a) An exception of accuracy

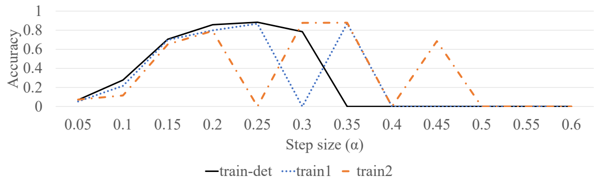


(b) Gradients changes for all layers

Figure 10: Debug the sudden drop of accuracy.



(a) Robustness of FGSM adversarial training.



(b) Robustness tests over different step sizes.

Figure 11: Debugging the sudden drop of accuracy.

and 89.71%. The adversarial training can achieve high robustness for majority of the runs. However, due to nondeterminism, the process may produce bad results, such as runs 4 and 6, where the accuracy is less than 15%. It is hence important to figure out the underlying reasons. Our tool enables such inspection. Specifically, since the perturbation in FGSM is updated with a step size α, we further evaluate the effect of different step sizes regarding the robustness of adversarial training. Figure 11b shows the evaluation accuracy of various step sizes for the vanilla trainings `train1` and `train2`, and the training with our tool `train-det`. With our tool, the results indicate that the step size significantly impacts the performance of the model, since step sizes larger than 0.35 do not improve the robustness. In contrast, such observation cannot be made in `train1` and `train2` due to their inherent nondeterminism. Note that they still fluctuate with large step-sizes.

The third case is how we use DETrain to debug numeric bugs. Due to the space limitations, it is moved to the Appendix A.

6.5 Threats to Validity

The nondeterminism we study is based on our study of TensorFlow and PyTorch. Other frameworks may have different sources of nondeterminism. In addition, our experiments only validate that we eliminate the nondeterminism exposed by the studied models. It is possible that there are unused functionalities in these frameworks that could lead to nondeterminism beyond our scope. Also, the architecture of models may also affect our tool's performance.

7 RELATED WORK

Developers of DL libraries e.g., PyTorch and Tensorflow, may manually handle non-determinism. PyTorch provides some guidance to make computation deterministic [22]. Basically, users need to seed random number generators manually, and set deterministic options if running on the CuDNN backend. In addition, it is on the developers' shoulder to make sure all user-defined random operations are deterministic. Tool *tfdeterminism* patches non-determinism sources, e.g., `bias_add`. It is integrated to TensorFlow version 2.1. However, the project is still under development, and provides limited solutions [3]. Researchers have employed a deterministic implementation of PyTorch to address reproducibility [18]. They also conducted a sensitivity analysis to expose the effect of non-determinism. Our tool works on existing frameworks. *Determined* is a framework for effective development of DL applications [10]. It provides some support to mitigate non-determinism. However, it mainly utilizes existing mechanisms provided by the underlying DL frameworks, which are limited.

Reproducing an entire experiment of DL application has been studied in recent years [20]. They mainly focuses on the attribution of nondeterminism and its effects. They neither provide systematical method to eliminate nondeterminism nor discuss checkpointing and replay. Record/replay techniques for conventional software systems usually snapshot the whole memory and all opened files. When users want to replay the program, the saved state is restored from the snapshot. Model training programs involve large dataset, thus the conventional methods introduce too much overheads. Moreover, they cannot eliminate nondeterminism caused by random numbers coupled with data parallelism. Existing checkpointing APIs simply save a copy of model parameters. With these parameters, one can hardly restore the program states since many critical information, e.g., the states of random generators, are missed.

8 CONCLUSION

Deterministic reproduction of a training process enables security auditing, debugging, and regression analysis. In this work, we investigate and model the sources of nondeterminism in DL training processes. A novel deterministic execution, checkpointing, and replay technique is then proposed. According to our evaluation results, the tool can faithfully reproduce a training run from not only the beginning, but also a checkpoint. We also show its effectiveness in problem diagnosis during DL training.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by IARPA TrojAI W911NF-19-S-0012, NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Nicholas Carlini and David A. Wagner. 2016. Towards Evaluating the Robustness of Neural Networks. *CoRR* abs/1608.04644 (2016). arXiv:1608.04644 http://arxiv.org/abs/1608.04644

[2] 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*. Ieee, 248–255.

[3] Duncan Riach. 2019. Deep Learning Determinism. https://pypi.org/project/tensorflow-determinism/.

[4] Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin. 2009. First-Aid: Surviving and Preventing Memory Management Bugs during Production Runs. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 159–172. https://doi.org/10.1145/1519065.1519083

[5] GroupLens. 2017. MovieLens datasets. https://movielens.org/.

[6] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access* 7 (2019), 47230–47244.

[7] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 193–208.

[8] Daniel Ho, Eric Liang, Ion Stoica, Pieter Abbeel, and Xi Chen. 2019. Population Based Augmentation: Efficient Learning of Augmentation Policy Schedules. In *ICML*.

[9] Hugging Face. 2019. Transformers. https://github.com/huggingface/transformers.

[10] Jennifer Villa, Yoav Zimmerman. 2018. Reproducibility in ML: why it matters and how to achieve it. https://determined.ai/blog/reproducibility-in-ml/.

[11] Keras. 2020. Callbacks API. https://keras.io/api/callbacks/.

[12] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. 2019. Fast AutoAugment. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[13] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[14] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning Attack on Neural Networks. In *25nd Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-221, 2018*. The Internet Society.

[15] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2019. Towards Deep Learning Models Resistant to Adversarial Attacks. arXiv:1706.06083 [stat.ML]

[16] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 693–708. https://doi.org/10.1145/3037697.3037751

[17] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[18] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. 2018. Deterministic implementations for reproducibility in deep reinforcement learning. *arXiv preprint arXiv:1809.05676* (2018).

[19] S. Narayanasamy, G. Pokam, and B. Calder. 2006. BugNet: Recording Application-Level Execution for Deterministic Replay Debugging. *IEEE Micro* 26, 1 (2006), 100–109.

[20] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. *Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance*. Association for Computing Machinery, New York, NY, USA, 771–783. https://doi.org/10.1145/3324884.3416545

[21] PyTorch. 2018. Deterministic cuDNN flag results in 2x speedup, how is this possible? https://tinyurl.com/y96yucrb.

[22] PyTorch. 2019. Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html.

[23] PyTorch. 2020. ImageNet training in PyTorch. https://github.com/pytorch/examples/tree/master/imagenet.

[24] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 235–248.

[25] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[26] Y. Shalabi, M. Yan, N. Honarmand, R. B. Lee, and J. Torrellas. 2018. Record-Replay Architecture as a General Security Framework. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 180–193.

[27] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 2004. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Boston, MA) *(ATEC '04)*. USENIX Association, USA, 3.

[28] Stack Overflow. 2015. Tensorflow NaN bug? https://stackoverflow.com/questions/33712178/tensorflow-nan-bug/.

[29] StackOverflow. 2018. TensorFlow: Are my logits in the right format for cross entropy function? https://stackoverflow.com/a/36086477.

[30] TensorFlow. 2019. The Model Garden for TensorFlow. https://github.com/tensorflow/models.

[31] Tensorflow. 2020. ModelCheckpoint. https://tinyurl.com/yxqnpr83.

[32] Tensorflow. 2022. ResNet in Tensorflow. https://github.com/tensorflow/models/tree/master/official/legacy/image_classification/resnet.

[33] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2017. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204* (2017).

[34] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/1294261.1294275

[35] M. Yan, Y. Shalabi, and J. Torrellas. 2016. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14.

[36] Yangyang Guo. 2020. A pytorch GPU implementation of NCF. https://github.com/guoyang9/NCF.

[37] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems* 30, 9 (2019), 2805–2824.

[38] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P. Xing, Laurent El Ghaoui, and Michael I. Jordan. 2019. Theoretically Principled Trade-off between Robustness and Accuracy. *CoRR* abs/1901.08573 (2019). arXiv:1901.08573 http://arxiv.org/abs/1901.08573

A CASE STUDY THREE: USE DETRAIN TO DEBUG NUMERICAL EXCEPTION

```
1  tf.global_seed = 123
2  def loss_fn(y_true, y_pred):
3      tmp = tf.math.log(y_pred)
4      tmp1 = tf.one_hot(y_true, num_classes) * tmp
5      loss = -tf.reduce_sum(tmp1)
6      tf.debugging.assert_all_finite(loss, "NAN Loss...")
7      return loss
8
9  # main body
10 model = Model()
11 model.loss = loss_fn
12
13 data = get_dataset()
14 data = data.map(λ img:random_flip(img))
15 model = Model()
16 for epoch in range(0...200):
17     for batchX, batchY in data:
18         model.fit(batchX, batchY)
```

Figure 12: Buggy code in a program reproduced from [28].

Fig. 12 shows a real-world bug [28] caused by passing 0 to the log function. This piece of code trains a model with the loss function `loss_fn`. The function starting at line 2 defines the cross entropy loss. Lines 10-18 are the main logic for training. The loss function takes as input the predictions generated by the model and the true labels, and computes the loss accordingly. It first computes the log values of predictions, then computes the cross entropy loss. Note that at line 6, the loss value is checked in an assertion statement. If the value is NaN, an exception is thrown. In the main logic, the developers first define the model and specify the loss function used to train the model. Then the dataset is loaded and augmented by randomly flipping some pixels in the images. Note that the API `data.map()` empowers the augmentation with data parallelism.

The bug in the example is at line 3. When the model's prediction `y_pred` evaluates to zero, the loss would be NaN. In a real-world scenario [28], the bug reveals itself after 50 minutes of training. Unfortunately, with the current support of existing frameworks, this bug can hardly be reproduced deterministically. On one hand, existing frameworks cannot deterministically transform the dataset, meaning that in each execution the model is trained on a dataset with potentially different augmentation. On the other hand, existing frameworks cannot faithfully reproduce the (failing) training process from a checkpoint. That is, the developers have to re-train the model from the beginning for many times to reproduce the bug.

With the support of DETrain, one can deterministically reproduce the bug. Moreover, the developer can restart the program from the most recent checkpoint to promptly locate and fix it.