HARP: Holistic Analysis for Refactoring Python-Based Analytics Programs

Weijie Zhou
wzhou9@ncsu.edu
North Carolina State University
Raleigh, NC

Yue Zhao
yzhao30@fb.com
Facebook
Menlo Park, CA

Guoqiang Zhang
gzhang9@ncsu.edu
North Carolina State University
Raleigh, NC

Xipeng Shen
xshen5@ncsu.edu
North Carolina State University
Raleigh, NC

ABSTRACT

Modern machine learning programs are often written in Python, with the main computations specified through calls to some highly optimized libraries (e.g., TensorFlow, PyTorch). How to maximize the computing efficiency of such programs is essential for many application domains, which has drawn lots of recent attention. This work points out a common limitation in existing efforts: they focus their views only on the static computation graphs specified by library APIs, but leave the influence from the hosting Python code largely unconsidered. The limitation often causes them to miss the big picture and hence many important optimization opportunities. This work proposes a new approach named HARP to address the problem. HARP enables holistic analysis that spans across computation graphs and their hosting Python code. HARP achieves it through a set of novel techniques: *analytics-conscious speculative analysis* to circumvent Python complexities, a unified representation *augmented computation graphs* to capture all dimensions of knowledge related with the holistic analysis, and *conditioned feedback mechanism* to allow risk-controlled aggressive analysis. Refactoring based on HARP gives 1.3–3X and 2.07X average speedups on a set of TensorFlow and PyTorch programs.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; *Data flow architectures*; • **Computing methodologies** → *Machine learning*.

KEYWORDS

machine learning program, computation graph, dynamic language, program analysis

ACM Reference Format:
Weijie Zhou, Yue Zhao, Guoqiang Zhang, and Xipeng Shen. 2020. HARP: Holistic Analysis for Refactoring Python-Based Analytics Programs. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380434

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380434

1 INTRODUCTION

For machine learning applications, computing efficiency is essential, for the growing scale of datasets and the needs for timely responses in various applications. A pattern common in today's analytics applications is *Python+X*, where, main computations are written with APIs of some libraries *X* (e.g., TensorFlow, PyTorch) while the host code is written in Python which connects all pieces together. We call them *Python-based analytics programs*.

Such programs have some common features. We draw on TensorFlow [1] as an example for explanation. Like some other packages, TensorFlow was initially introduced for a specific scope (Deep Learning), but then evolved for a broader scope (Analytics and Scientific Computing). At its core, TensorFlow builds on the dataflow programming model. In developing a TensorFlow program, a developer writes code by calling TensorFlow APIs to specify the intended computations, and makes a call to the TensorFlow runtime. The call triggers the execution of those APIs, which constructs a computation graph; the TensorFlow runtime optimizes the graph, and then executes it. This approach represents one of the popular programming paradigms used by modern machine learning and analytics frameworks.

The recent years have seen a number of efforts for enhancing the performance of Python-based analytics programs. For example, the XLA project [18] and the R-Stream-TF project [31] try to improve the runtime performance of TensorFlow by utilizing compiler techniques to transform dataflow graphs (e.g., fusing multiple operations).

However, despite the many efforts, a large room for performance improvement is left elusive for current optimizers to harvest. We believe that one of the fundamental reasons is that current efforts have all focused on the dataflow graphs embodied by the invocations of library APIs in a program. Although dataflow graphs usually capture the core computations of an analytics program, limiting the view on them may lose the big picture that the host code provides, and hence many large-scoped optimization opportunities.

Listing 1 offers an example. This codelet is part of the core computations in Deep Dictionary Learning [24]. The first five lines of the code specify the core computations and hence the structure of

the dataflow graph. Line 1 defines A as a variable as its value needs to be updated through the execution; Lines 2 and 3 define D and X as place holders for they will hold the input values; Line 4 specifies the formula for calculating the new value of A through a series of calls to the high-level TensorFlow mathematics functions; Line 5 specifies an update to A with the new value. Line 6 is a Python loop statement, and Line 7 calls TensorFlow API "sess.run" (a blocking API) to invoke TensorFlow runtime to actually construct the dataflow graph and execute it; it passes the actual parameters $D\_$ and $X\_$ to the two place holders.

The following pseudo-code shows what the codelet implements:

$$
\begin{aligned}
&for\ i < Iter: \\
&\quad A = D(X - D^T A)
\end{aligned}
$$

where each of the capitalized letters represents a multidimensional array (called a tensor), and D and X are input tensors and remain constant across the loop iterations.

The implementation by the codelet in Listing 1 contains some redundant computations, which can be seen if we expand the mathematical formula in the pseudo-code to $DX - DD^T A$. Because terms DX and DD^T are invariant across the loop iterations, they can be hoisted outside the loop, computed once and reused for all iterations as shown as follows:

$$
\begin{aligned}
&t_1 = DX \\
&t_2 = DD^T \\
&for\ i < Iter: \\
&\quad A = t_1 - t_2 A
\end{aligned}
$$

Even though the redundant computations are not hard to see at the pseudo-code level, it cannot be recognized or optimized by either TensorFlow or any of the previously proposed optimizers designed for TensorFlow programs. Moreover, even if we rewrite the code to better expose the redundancy as in Listing 2, prior optimizers still cannot detect the redundant computations.

The reason is that all these tools focus only on the dataflow graph composed through the TensorFlow APIs, while the redundancy comes from the interplay between those TensorFlow API calls and the other Python code that hosts those calls. Take Listing 1 for instance, without analyzing the host Python code, it is impossible to tell that D and X are invariant across the `for` loop and hence DX and DD^T are also loop invariants, as the computation graph does not capture the loop in the host code and its relation with the data in the graph. On the other hand, general Python code optimizers cannot find out the redundancy either as they do not understand the necessary semantics of the TensorFlow APIs. Other examples include partially repeated computations involved two separate API calls, the mis-use of computation graph construction APIs as computation APIs in a loop and causing millions of graph nodes to be generated unnecessarily (See Section 4.3).

This work proposes a new approach named HARP (Holistic Analysis for Refactoring Python-Based Analytics Programs) to address the problem. We next gives an overview of HARP, the main challenges and our contributions.

Listing 1: Example from Deep Dictionary Learning ("tf" for the namespace of TensorFlow)

```
1 A = tf.Variable(tf.zeros(shape=[N, N]), dtype=tf.float32)
2 D = tf.placeholder(shape=[N, N], dtype=tf.float32)
3 X = tf.placeholder(shape=[N, N], dtype=tf.float32)
4 R = tf.matmul(D, tf.subtract(X, tf.matmul(tf.transpose(D)
      , A)))
5 L = tf.assign(A, R)
6 for i in range(Iter):
7     result = sess.run(L, feed_dict={D: D_, X: X_})
```

Listing 2: Listing 1 in a Different Form

```
1   ...
2   t1 = tf.matmul(D, X)
3   t2 = tf.matmul(D, tf.transpose(D))
4   R = tf.substract(t1, tf.matmul(t2, A))
5   L = tf.assign(A, R)
6   for i in range(Iter):
7       result = sess.run(L, feed_dict={D: D_, X: X_})
```

2 OVERVIEW OF HARP

To overcome the aforementioned limitation of existing tools, an initial option we considered was to create an automatic static code optimizer with a holistic view. It is however impractical: Python is a dynamically-typed language, the complexities of which (listed later) form some major barriers for static optimizers to work effectively. A pure dynamic optimizer on the other hand is complicated to develop and causes runtime overhead and delays.

The strategy we finally took is to create a refactoring assistant, which provides suggestions for developers to use in refactoring the code. This strategy allows aggressive analysis on incomplete information. Even though the suggestions may not be always correct, as the tool provides enough feedback on the assumptions it uses, the developers can avoid the risks while taking advantage of the often correct suggestions. HARP is the first tool that enables holistic analysis that spans across dataflow graphs and their hosting Python code. HARP achieves it through several novel features:

(1) Speculative analysis. HARP relies heavily on static analysis. As a dynamically-typed scripting language, Python poses many difficulties to static analysis, such as unknown types, operator overloading, dynamic dispatch of higher-order functions, and so on. HARP circumvents the complexities with two key insights: Many Python complexities rarely appear in analytics programs or can be ignored; for those that do matter, they can often be treated successfully through speculations on common patterns in analytics applications. HARP materializes the insights through *speculative analysis*.

(2) Uniform representation. Holistic analysis requires a coherent way with a uniform structure to represent the computations and relations attained from both the host and the computation graph. The representation must be inclusive in the sense that it must contain necessary info from both the host and the library—including necessary high-level semantics (e.g., side effects, expected tensor type) of APIs. It, at the same time, must be amenable for automatic inferences for optimization opportunities. HARP introduces *augmented computation graphs*, a uniform representation that augments computation graphs with relations attained from the host code and some light annotations of library APIs. It uses Datalog as the media to create the coherent representation for holistic analysis.

(3) Informative and extensible interface. HARP employs a Datalog-based interface such that code analysis modules can be simply

expressed in Datalog rules in a declarative manner. Through the interface, we equip HARP with a set of predefined rules for detecting common inefficiencies on the augmented computation graphs. The interface also allows users to easily extend HARP with extra rules. Meanwhile, we equip HARP with a *conditioned feedback mechanism*, through which, HARP gives users not only suggestions for code refactoring but also the assumptions it holds in its speculations. This feature helps control the risks of the speculative analysis, and allows HARP to offer likely useful suggestions despite language complexities.

To evaluate the efficacy of HARP, we implement it as a plugin of PyCharm based on IntelliJ[1] and develop the support for TensorFlow and PyTorch. HARP is effective in finding optimization opportunities that are elusive to prior techniques. Code refactoring based on the findings yield 1.3-3X performance improvement on a GPU machine. We further conduct a user study, which helps confirm the productivity benefits of this refactoring tool.

We do not claim using Datalog for program analysis as our contribution. Many previous papers have applied Datalog for program analysis and shown promising productivity [2, 14, 37–39]. The key contributions of this work are four-fold:

- It points out the limited view of existing tools as a fundamental barrier to harvest important opportunities for refactoring modern machine learning applications for performance.
- It provides several insights important for circumventing language complexities in machine learning applications, and proposes ways to leverage common patterns in machine learning to enable effective speculative analysis.
- It develops *augmented computation graphs* as unified way to host all dimensions of knowledge related with the holistic analysis.
- It creates HARP, the first holistic analysis-based tool for machine learning code refactoring, and demonstrates its effectiveness in a range of applications.

The current implementation of HARP focuses on supporting TensorFlow and PyTorch [29] programs for their increasing popularity, and also for their representatives of two paradigms in computation graph construction: TensorFlow builds static computation graphs, while PyTorch defines dynamic graph. We next provide some background knowledge and then describe each part of HARP in detail.

3 BACKGROUND

This section presents background knowledge on TensorFlow, PyTorch, and Datalog.

3.1 TensorFlow

A TensorFlow program works in a "define-and-execute" way [13]. Its execution first creates a computation graph which then gets optimized and executed by the runtime. High level languages such as Python are often used as the host language in defining the computations, and low-level languages such as C-family languages are often used to implement the runtime system for the purpose of performance efficiency. A TensorFlow program can be regarded as a mix of two program components: The first part, called "host program", contains the computation that is directly executed by the front-end language. The second part is the computation represented by the "computation graph".

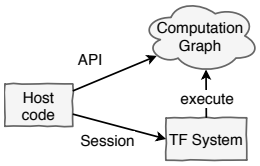

Figure 1: High-level relations among host code, TF system, and computation graphs.

Computation graphs. In TensorFlow, the computation graph is a directed graph composed of a set of nodes and edges:

- Nodes instantiates operations (e.g., "matrix multiply" or "sigmoid"). Each operation can have zero or more inputs and zero or more outputs.
  These operations get dispatched to devices (CPU, GPU) by the runtime.
- Edges represent the values communicated between operations. The most common types of values in TensorFlow are *tensors*, which are N-dimensional arrays. Their elements can have one of the primitive types (e.g., `int32`, `float32`, or `string`). Tensors on edges are stateless and immutable.
- There are some special nodes and edges. For example, a `tf.Variable` creates a variable, which represents stateful value and is mutable. It is represented as a node in the graph carrying the variable name as its label. In addition, there are `control dependency` edges which indicate controls of the execution order of operations.

Host program and graph execution. In a typical TensorFlow program, the responsibilities of the host program include preparing data for the computation, hosting the API calls that define the computation graph, and controlling its executions.

The host program interacts with computation graphs and the underlying TensorFlow (TF) system through the `session` API. It calls `session.run` which prompts the TF system to execute the computation graph. The call may specify a set of output nodes whose values are to be *fetched*, and a set of input tensors to be *fed* into the graph. Figure 1 illustrates the relations.

A computation graph can be executed multiple times. It is important to note that `tensors` do not survive across one execution of the graph (e.g., one session.run); the memory holding them is allocated and reclaimed automatically. In contrast, values of *variable* nodes persist across executions.

3.2 PyTorch

PyTorch is another popular machine learning framework. Similar to the TensorFlow, the PyTorch program can also be regarded as a mix of the Python host program and the underlying computation graph of operations, which will be dispatched to runtime kernels in the execution. However, unlike TensorFlow, the PyTorch builds the computation graph dynamically. In other words, the PyTorch

[1] http://www.jetbrains.org

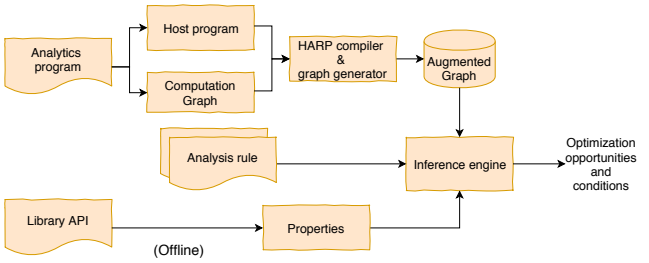

Figure 2: The overall system diagram.

constructs the computation graph on-the-fly so the computation graph can change every time it is executed.

3.3 Datalog

Datalog is a popular logic programming language based on the first order logic. In Datalog, program logic is expressed in terms of relations, represented as facts and rules, and computations are in queries over these relations. Two basic constructs are *term* and *atom*. A term is either an entity (constant) or a variable[2]. An *atom* is a predicate with a list of terms as arguments. It is in form of $p(X_1, X_2, \ldots, X_n)$, where p is a predicate and X_i are terms. A *ground atom* or *fact* is a predicate with only constant arguments. Many facts together form a Datalog database.

Datalog rules express logical inferences, in the following form:

$$A \text{ :- } B_1, B_2, \ldots, B_n.$$

which reads "B_1 and B_2 and ... and B_n together imply A", where each symbol is an atom.

A Datalog program is a collection of rules. When it runs on a fact database, it infers a new set of facts by applying the rules to the known facts. Datalog can define recursive relations and recursive queries naturally and is declarative.

4 THE HARP SOLUTION

This section presents HARP in detail. Figure 2 outlines the overall structure of HARP. HARP code analysis extracts the relevant information from the host code and the computation graphs, and represents them in a unified format, *augmented computation graph*, written in Datalog atoms. Meanwhile, through some light annotations, we equip HARP with the knowledge on the high-level properties of the library APIs, including the side-effects of APIs, argument types and returning types. With all the relevant information captured and represented in an analyzable manner, for a rule (predefined in HARP or given by users) describing code inefficiency patterns, the Datalog inference engine can identify optimization opportunities that span across boundaries between host code and API calls. HARP is equipped with a list of predefined rules on code inefficiency, which are derived through our analysis of 112 real-world analytics applications, listed in Table 1 and elaborated in Section 4.3. We next explain HARP and how it addresses the main challenges in enabling holistic analysis.

[2]By conversion, lowercase for constants and uppercase for variables.

Listing 3: An example machine learning codelet

```
1 def dfdx(x, param, f):
2     z = f(x, param)
3     z.backward([torch.ones_like(z)])
4     dfdx_v = x.grad
5     return dfdx_v
```

4.1 Overcoming Language Complexities

The first challenge this work encounters is Python complexities. As an important step outlined in Figure 2, the *HARP analyzer* must capture the important interactions between the host Python code and the computation graphs. As a dynamically-typed scripting language, Python poses many difficulties to the static analysis, which are summarized in Table 2. These complexities cause unknown types and undecidable operations or function calls.

Listing 3 shows such an example. The function `dfdx` calculates the partial derivative of a function with regard to the input `x`. Due to the dynamic feature of Python, it is hard for static analysis to precisely determine the types of its input parameters.

HARP circumvents the difficulties by leveraging insights obtained specially on analytics applications. The first insight is that many Python complexities rarely appear in analytics programs or can be ignored. Table 2 reports the number of appearances of each type of complexity in 112 real-world analytics applications that we have surveyed in two GitHub collections [3]. These two collections contain some TensorFlow and PyTorch programs representing a variety of tasks in machine learning, ranging from natural language processing to image processing, object detection, and so on. In the survey, we ran our static Python code analyzer (sec 4.2.2) on each of the programs, which reports places that give static analysis a hard time. We then manually examined those cases. Among the 112 applications, the most frequent complexity is dynamic type changes, but the frequency is still only 14.

The second insight is that for those complexities that do matter, they can often be treated successfully through speculations.

The speculations are based on some common patterns observed in analytics applications. Analytics applications, due to the common features of the domain, exhibit some patterns. For instance, in PyTorch code, the type of a variable that has a member function "grad" or "backward" is usually a tensor with all floating-point values. That pattern can then help speculate on the types of variables x and z in Listing 3. Although using names for speculation is in general not fully reliable, we observed that doing that for some common functions in machine learning domain gives correct speculations in most of times. Table 3 lists some of the patterns we attained through our studies on our collection of machine learning applications. Listing 3 belongs to the second pattern in Table 3.

When encountering a complexity in its static analysis of a given Python program, HARP resorts to the common patterns. If the conditions match, HARP resolves the complexity speculatively.

Such speculations allow HARP to carry on its code analysis even in the presence of the complexities. That makes it possible to provide refactoring suggestions that are likely (not guaranteed) to be valid and beneficial. Speculations could be wrong; the developer

[3]https://github.com/jtoy/awesome-tensorflow; https://github.com/ritchieng/the-incredible-pytorch

Table 1: Some Inefficiency Patterns Predefined in HARP

No.	Inefficiency	Code Patterns	Potential Optimization's
1	Mistake the computation construction for computation execution (TensorFlow)	• Use computation graph construction APIs inside loop. • The APIs are primitive operators such as `tf.add`, `tf.mul`, etc.	Do the computation in the `Session`.
2	Loop redundancy	• Loop invariant computation inside loop	Hoist the loop invariant out of the loop
3	Common subgraph computation	• Two consecutive executed computation graphs have common intermediate nodes. • The value of the common nodes are invariant across the executions.	Cache the last common nodes and reuse the value.
4	Plain *for* loop for computation (TensorFlow)	• Use the plain Python loop for building computation graph. • The code inside the loop only reference nodes in the computation graph but not other data on the host side.	Replace the plain Python loop with `tf.while_loop`
5	Miss performance tuning for static graph (PyTorch)	• The dynamically constructed computation graph remains the same across loops: no conditional branch or dynamic dispatch. • Size of the input data is not changed. • Iteration number is non-trivial (>5).	Turn on the CUDA specific optimization tuning such as `torch.backends.cudnn`
6	Scalar computation that can be vectorized	• Simple *for* loop contains straight line code with neither function calls nor branches. • Computation on vector data type (list of number, array, etc). • No vector dependence.	Use array or tensor type and their vectorized operations.

Table 2: Python language complexities for static analysis and appearing frequencies in 112 machine learning applications.

Python Language Complexities	# programs with the complexity
Data dependent conditional branch	3
Static unknown function dispatch	4
Higher-order function	10
Customized operator overload	2
Dynamic attribute setting	4
Dynamic type changes	14
Dynamic-typed container	12

Table 3: Some of the machine learning patterns leveraged for speculative analysis.

No.	Class of patterns
1	Access to an object through its attribute, such as *foo.bar*, or an index, such as *foo[bar]*, is speculated as side effect free.
2	The type of an object can be speculated based on the names of its fields and methods visited locally if the names are common machine learning names (e.g., those in Figure 3).
3	If a Python list is (speculatively) converted to a tensor, it is speculated that the elements in the list share the same type.
4	If all branches of a condition statement return variables with the same type, they speculatively share the same type.
5	I/O operations are normally used for loading data and logging, thus they are speculated as having no unexpected side effects.

```
torch.utils.data.Dataset(), transforms.CenterCrop(),
transforms.Normalize(), torch.from_numpy(t),
transforms.functional.adjust_brightness(img…),
torch.ones_like(a), torch.add(a,b), Tensor[:,:,...],
Tensor.abs(), Tensor.add(b), Tensor.min(), Tensor.copy(),
Tensor.dim(), Tensor.size(), Tensor.permute(),
Tensor.reshape(), Tensor.type_as(), Tensor.float(),
Tensor.detach(), Tensor.cuda(),
Tensor.unsqueeze_() ...
```

Figure 3: Some common function names used in HARP as hints for speculative analysis.

needs to make the final judgment. To assist the developers in the process, HARP records all speculations, and reports them when it provides the corresponding refactoring suggestions, as Section 4.4 details.

4.2 Unified Representation through Augmented Computation Graphs

For holistic analysis, it is preferred that all the needed information of the program is represented in an analyzable form. It implies three questions: (1) what information to get, (2) how to get it, and (3) how to represent it. HARP answers these questions by following several design principles:

- First, the representation should be amenable for software inference, but also easy to understand for humans. The representation can then be used for not only program analysis, but also as a kind of documentation.
- Second, the set of information to collect from the program are intended to be used in a variety of analysis; hence, their definitions should be general rather than tailored to some particular analysis.
- Third, the representation should also be extensible. Thus, specification for new properties and other sources of information can be added easily.

We next explain the solutions from HARP in detail.

4.2.1 Info to Attain: Semantic Definitions. The relevant information for holistic analysis comes from both the host Python code and the computation graphs. For simplicity of explanation, we use "semantics" to refer to all[4]. Both host code and the computation graph carry many-fold semantics, defining the set that is most relevant to large-scoped inefficiencies is the key.

Semantics from Computation Graph. As mentioned in Section 3, a computation graph consists of two types of objects, nodes and edges. Each node represents an operation that consumes or produces values. Each edge represents the values that are output from, or input to a node, namely, the values that flow along the edge. We use TensorFlow as the example to explain the set of semantics HARP captures from nodes and edges.

[4]The meaning goes beyond traditional language semantics, referring to any properties or knowledge about a programming language construct.

For a node (TF operation), HARP captures four-fold semantics (1) *control_inputs*, referring to the set of operations on which this operation has a control dependency; (2) *inputs*, which are the list of tensor inputs of the operation; (3) *outputs*, which are the list of output tensors of the operation; and (4) *type* of computation, such as `Add`, `MatMul`. HARP contains the set of common computation types, including those supported in ONNX [8], a standard for representing deep learning models; more can be easily added.

For an edge (TF value), HARP captures some important properties: *kind*, *dtype*, *shape*, *constant or variable*. The first, *kind*, is about which kind of edge it is. We distinguish two different kinds of edges based on their different purposes [1]:

- *Tensor edges* convey the immutable tensors of the data. In TensorFlow, all data are modeled as tensors and there is no need to distinguish tensors or scalars.
- *Control edges* do not carry values. They are special edges used to constrain the order of execution.

The second property, *dtype*, is the type of data carried on the edge, which can be any of the supported data types in TensorFlow. The third property, *shape*, records the shape of the tensor, which is a list $[D_0, D_1, \ldots, D_{d-1}]$ storing the size of each dimension of the tensor. The final property indicates whether the tensor on the edge is *constant* or mutable.

Semantics from Host Code. For TensorFlow, HARP captures the following semantics that closely relate with the core computation:

(1) The operations that invoke TensorFlow APIs to define the computation graph. For example, the statement `c = tf.add(a, b)` in the host program defines an *Add* operation in the computation graph. These APIs are high order functions that return operations. We convert them into "operation creation" nodes whose outputs are special edges representing "operations".

(2) The invocation of the `session` API which prompts graph executions. This is also the interface for the host-computation graph interaction.

(3) The control flow of the host program.

(4) The processing of data that is to be consumed by the computation graph, which includes calls to other library APIs (e.g., Numpy).

4.2.2 Collecting Semantics through HARP Compiler. Deriving the semantics is done through our *HARP compiler* and computation graph dumping. The compiler is developed based on Jedi [33], a static analyzer of Python programs.

Computation graph. The step to get the semantics of the computation graph is as follows. For TensorFlow programs, the compiler inserts a graph dumping call (and a follow-up exit) at the place in the program where the computation graph is built (e.g., invocation of "run()" of a tf.Session object), and then runs the program on a provided sample input. The computation graph is then dumped in a format easy to parse. For most programs, the automatic method can succeed. In rare cases when the code does not follow the common patterns, HARP provides feedback and users can insert the dumping call (as part of HARP interface). The exported graph is in a format called `GraphDef`, which can be easily parsed to get the structure relations and properties of nodes and edges. For PyTorch programs, we use an approach similar to existing tools (AutoGraph [25] and JANUS [11]) to create the computation graphs.

Host program. To get the semantics from the host program, both data flow analysis and control flow analysis are necessary. Two properties of machine learning applications ease the process. First, the set of data that requires attention is limited. Since the data is fed into the graph through the `session` API, we only need to focus on data in the feed list. Thus, the domain of the dataflow analysis is largely narrowed down. Second, in most of the deep learning or machine learning programs, the data has a straightforward workflow. For example, a typical flow of the input data is that it starts from being read from the external storage or being generated from other components of the program, and then it is converted to some tensor variables. The variable is then provided to the computation graph as the input data. So the data of interest often exhibits a linear control flow. HARP lowers the representation of some control flow structures by following existing practice [7] (e.g, lowering `if` statement to `switch` and `merge` primitives). HARP compiler circumvents Python language complexities through the speculative analysis described in the earlier section. It records the assumptions it uses when applying a speculation, which will be used later as part of the feedback to users.

4.2.3 Representing Semantics: Augmented Computation Graph. We design *augmented computation graph* for HARP to use to seamlessly integrate the host code semantics with those of the computation graph. To make the representation amenable for existing logic inference engines to use, we represent all semantics as Datalog atoms using a single vocabulary.

Augmented Computation Graph. Extensions are added to the default computation graph to accommodate each category of semantics from the host code.

- HARP uses inter-procedural graph to model the interplay between host code and the computation graph. The inputs to a computation graph are treated as function parameters and the outputs as function returns. The analysis can hence take advantage of standard inter-procedural analysis.
- For the control flow structure in the host program, the most important one is the *loop* structure, which creates a local scope containing a set of computations. To encode its semantics, a loop is modeled as a special operation node in the augmented graph. For any computation inside the loop, there is a control edge from the loop node to the node representing the computation. If a data is loop variant, there is an extra tensor edge from the loop operation to it. Edges go from a loop node only to operations in the host program, which then may connect to the operations in the computation graph. It is worth noting that, the relations between the `loop` operation and the other nodes reflect dependencies, rather than traditional structures in the abstract syntax tree (AST) of the host program.
- For the influence to host data dependencies and control flows from calls to other library functions (e.g., `Numpy`), HARP gets the semantics through library annotations. Particularly, we annotation library APIs such as *type signature*, *side-effect*.

Listing 4: Specifications for edges

```
edge(<edge_id>, kind, <kind>).
edge(<edge_id>, dtype, <data_type>).
edge(<edge_id>, shape, <tensor_shape>).
```

Listing 5: API Misusage Example from StackOverflow

```
x = tf.Variable(...)
...
for _ in range(1e6):
    x = x + 1
...
sess.run(...)
```

Datalog Atoms. The augmented graph is represented in HARP in the form of Datalog atoms. In our design, each atom only describes one attribute of the object. This is for the flexibility and extensibility. For example, if new attributes for the tensor is needed for new analysis, they can be added to the semantic schema without breaking the existing specifications and rules.

Listing 4 shows the main kinds of atoms for edges (TF values), corresponding to the three kinds of semantics defined for edges. The atoms of nodes are in similar forms but with different keywords for different semantics.

4.3 Inefficiency Detection Rules and Engines

With the augmented computation graph written in Datalog atoms, program analysis algorithms can be written in Datalog rules. This design has several benefits. First, the declarative interface provided by the logic programming lowers the bar for writing complicated algorithms than the imperative interface of traditional compilers does. Second, the semantics of machine learning programs often center around the graph structure. The relational and recursive nature of logic programming languages makes it a good fit for the graph-based analysis. For example, in Datalog, the template of recursive algorithms are as simple as follows:

```
recursive_relation(EdgeFrom, EdgeTo) :- direct_relation(
    EdgeFrom, EdgeTo).
recursive_relation(EdgeFrom, EdgeTo) :- direct_relation(
    EdgeFrom, EdgeMid), recursive_relation(EdgeMid,
    EdgeTo).
```

The template queries the relation between every pair of edges. The template simulates the process of computing the transitive closure of the graph. It starts from the existing facts (the direct relation) and inference other facts through recursive propagation. Rules for nodes can be defined similarly.

Through analysis of 112 real-world analytics programs [5], we summarized a list of code inefficiency patterns in Table 1. We next illustrate the simplicity in defining corresponding rules for detecting inefficiency on the augmented graphs.

Example 1: API sanitizing. The first inefficiency in Table 1 is misuse of the library APIs on constructing computation graphs. The code pattern is that those APIs are used inside a loop, causing many nodes to be created unnecessarily, as Listing 5 shows. HARP has

[5]https://github.com/jtoy/awesome-tensorflow; https://github.com/ritchieng/the-incredible-pytorch

Listing 6: RBM codelet

```
alpha = 0.5
a = tf.placeholder("float", [100])
b = tf.placeholder("float", [784])
W = tf.Variable(tf.random([784, 100]))
x = tf.placeholder("float", [None, 784])

h0 = tf.nn.sigmoid(tf.matmul(x, W))
h1 = tf.nn.sigmoid(tf.matmul(h0, tf.transpose(W)) + b)
v1 = tf.nn.sigmoid(tf.matmul(h1, W) + a)

y1 = tf.reduce_mean(x - v1, 0)
y2 = tf.reduce_mean(h1 - v1, 0)

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())
  for _ in range(10):
    for start, end in indices:
      X, A, B = data_gen(start:end)
      _y1 = sess.run(y1, feed_dict={x: X, a: A, b: B})
      A = alpha * A
      _y2 = sess.run(y2, feed_dict={x: X, a: A, b: B})
```

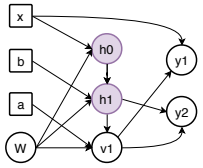

Figure 4: The simplified dataflow graph for Listing 6. The square nodes represent data variables and circle nodes represent operations. The labels inside the operation nodes indicate their output names.

rules (called "API sanitizing rules") to detect such API misuses. The rules are simple, thanks to the augmentation graph representation and the declarative interface HARP uses. The code below expresses the pattern of "including operation creating APIs inside a loop":

```
op(O, type, "OpCreation"), op(O, insideLoop, "loopID")?
```

where `control_edge` indicates the operation is inside a loop.

Another use case of the sanitizing rule is to help shape inference. In TensorFlow, the shapes of some tensors can be unspecified, which are *dynamic shapes*. At runtime, the TensorFlow system would need to infer the shape based on the input data, and then propagate the info to other tensors that depend on these input tensors. Dynamic shapes are hurdles for TensorFlow to simplify computation graphs or enable efficient memory allocations.

In many of the cases with *dynamic shapes*, however, the input data can actually be determined from the host code (e.g., deep learning with a fixed batch size). But as TensorFlow compilers focus on computation graphs only, it cannot do the shape inference in advance. The holistic analysis by HARP addresses the limitation and detects unnecessary dynamic shapes.

Example 2: Detecting computation redundancy. A more interesting use of the HARP framework is to hunt for the computation redundancies (patterns two and three in Table 1) that are elusive to the existing tools and subtle to see by programmers. We demonstrate it on the example in Listing 2 of Section 1. Figure 5 shows the augmented computation graph including the relevant operations

Listing 7: Dependency checking rules

```
direct_dep(E1, E2) :- input_edge(Node, E1), output_edge(
    Node, E2).
dep(E1, E2) :- direct_dep(E1, E2).
dep(E1, E2) :- direct_dep(E1, X), dep(X, E2).
```

Listing 8: Shape consistency checking rules

```
shape(E, S) :- feed_in(E), S == shape_check(E).
shape(E, S) :- out_edge(Node, E), findall(IE, input_edge(
    Node, IE), IEs), shape_check(Node, IEs, S).
```

from the host program. It captures all the constant information and the scope of the loop. Through constant propagation, HARP infers that `t1` and `t2` are invariant across the loop and thus they can be hoisted. The suggested code refactoring is to change the data type of `t1` and `t2` to `tf.Variable`, making their values survive across the loop iterations and get reused.

Another example is to detect common sub-computation redundancy as the one shown in Listing 6. Figure 4 shows a simplified dataflow graph corresponding to the RBM codelet. The outputs `_y1` and `_y2` both need the intermediate value `h0`, `h1` and `v1`. Node `a` affects `v1`, but `h0, h1` have the same values in the computations of `_y1` and `_y2` according to the Python code in Listing 6. Therefore, potentially, the values of `h0` and `h1` produced during the `session.run` for `_y1` could be reused for the `session.run` that produces `_y2`. To detect such common sub-computation redundancy, rules are written to find intermediate values common to multiple fetched values, such that they can be potentially cached and reused. The core of the detection algorithm is to analyze the dependencies of the values. The data dependency atoms are shown in Listing 7.

One can query for finding common dependencies. For example, if we have two fetched values $o1$ and $o2$, we can use the following query to find operations that both $o1$ and $o2$ depend on:

```
common_dep(X, o1, o2) :- dep(X, o1), dep(X, o2).
```

By adding an extra check, whether $o1$ and $o2$ belong to two different sessions, one can then find cases worth caching the value of X in one session and reusing it in the other session.

There are some other rules in HARP, including *constant propagation*, which finds all constant variables, *finding for-loop for victimization's*, which finds for-loops in Python code that operate on container objects likely containing elements of the same type and unlikely having loop-carried dependency, *static graph identification*, which identifies graphs that remain unchanged across iterations, *useless tensor detection*, which finds tensors that are not used. These analyses all lead to some potential optimizations. For *static graph identification*, for instance, the knowledge would allow the inclusion of some special flags (`torch.backends.cudnn.benchmark`) which would allow PyTorch engine to apply some aggressive optimizations to such graphs to improve the performance.

Inference engine. The inference engine of HARP is the Datalog inference engine with an extension to allow the invocations of external side effect-free imperative functions. The `shape_check` function in Listing 8 is such an example. It checks the shape consistency which means if an operator node has an outgoing edge

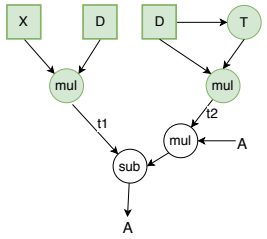

Figure 5: Constant propagation on augmented computation graph identifies loop invariants (the shadowed nodes).

(output), its shape should comply with the expected shape of the operation and the inputs. For example, for matrix multiplication of A and B, the output should have a shape of Rows(A) X Columns(B). As the check requires shape calculations, it is simple to write in imperative programming language but cumbersome in pure Datalog predicates. By extending Datalog predicates with imperative functions, HARP creates conveniences for the development of inference rules. Caution must be taken to make sure that the extension does not break the safety of the Datalog deduction. In the `shape_check` example, since the function has no side-effect on the Datalog atoms, it is regarded as a safe extension.

4.4 Informative Feedback

When the analysis by HARP involves speculations, the results could be wrong. HARP provides *conditioned feedback* to keep the risks controlled. The principle is to report also the assumptions HARP uses in its speculations when it offers refactoring suggestions. Figure 6 provides an example output from HARP, after it is integrated into the IDE PyCharm through IntelliJ. In this example, HARP identifies TensorFlow computation redundancies among the lines that invoke `sess.run()`. Then the IDE plugin highlights involved code lines. When the mouse hovers over the highlighted lines, a tooltip prompts the suggestion for the optimization, as well as the assumptions that HARP takes in providing the suggestion. The feedback needs correlate graph nodes and edges with source code of the program. To facilitate it, HARP compiler conducts a preprocessing of the given program by rewriting it to add unique name to each computation graph construction statement. For example, `c = tf.add(a, b)`, becomes `c = tf.add(a, b, name= "id")`, where `"id"` is a unique name managed by HARP. Then, the default TensorFlow backend automatically assigns that unique name to the corresponding node in the computation graph.

5 EVALUATION

In this section, we report the evaluations of the efficacy of the proposed HARP approach. We demonstrate how it can help diagnose TensorFlow and PyTorch programs to find subtle pitfalls and optimization opportunities that are elusive to existing tools. We report the significant performance improvement of the corresponding refactored code, as well as the effectiveness and benefits of the speculative analysis. We further report a user study on the usability of HARP as a refactoring tool for programmers.

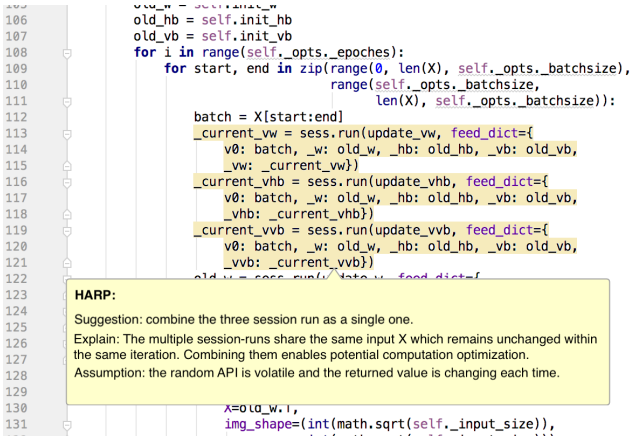

Figure 6: Example feedback from HARP on a Restricted Boltzmann machine (RBM) implementation.

Table 4: Hardware platforms used in the experiments.

	Intel® CPU	NVIDIA® GPU
CPU	Xeon E5-1607	GeForce GTX TITAN X
Freq.	3.00 GHz	1.08 GHz
Cores	4	3072
Memory	16GB DDR3 1.9 GHz	12GB GDDR5 3.5 GHz
Memory Bandwidth	34.1 GB/s	168 GB/s
OS/Driver	Ubuntu 18.04	CUDA 9.0
Compiler	GCC (6.2)	NVCC (9.0)

Table 5: Benchmarks and Datasets.

Benchmark	Application	Dataset
dict-learn	Dictionary Learning [24]	Random generated (size: 1000X1000)
rbm [22]	Restricted Boltzmann Machine [10]	MNIST [19]
char-rnn [20]	Recurrent Neural Networks [16]	Shakespeare work [17]
style-transfer [30]	Neural Style Transfer [9]	VGG-16 model, input size of 224X224
deep-q-net [15]	Deep Q-Network [35]	OpenAI Gym [3]
seq2seq	Variable Length RNN	Blog Authorship Corpus [32]
cycleGAN [40]	Image to image translation	Cityscapes dataset [5]
cholesky	Cholesky Decomposition	Random generated (size: 10000× 10× 10)

5.1 Experimental Setup

The hardware specifications for the evaluation platform are detailed in Table 4. The benchmarks are tested with GPU acceleration. The implementation of HARP is based on the Python static analysis tool Jedi [33] and the inference engine is written based on *Flix* [23]. HARP is integrated into the IDE PyCharm through IntelliJ to serve for code refactoring. Table 5 lists all the benchmarks. These benchmarks are outside the 112 surveyed benchmarks for identifying the speculation and optimization patterns. We collect these benchmarks from published work or real-world TensorFlow and PyTorch programs. They implement a range of machine learning algorithms for machine learning. Table 5 also shows the input dataset for each benchmark. These benchmarks are based on Python 3.6, TensorFlow 1.5, and PyTorch 0.4.1. In our experiment, each benchmark was executed multiple times; we saw only marginal fluctuations in the execution time, and hence reported the mean value.

Table 6: Potential efficiency problems reported by HARP.

Benchmark	Inefficiency reported by HARP (pattern numbers defined in Table 1)	True positive
dict-learn	2. Loop redundancy	Y
	4. Plain *for* loop for computation	N
rbm	3. Common subgraph computation	Y
char-rnn	4. Plain *for* loop for computation	Y
	6. Scalar computation that can be vectorized: Case I	N
	6. Scalar computation that can be vectorized: Case II	Y
style-transfer	3. common subgraph computation	Y
	4. Plain *for* loop for computation	Y
deep-q-net	3.Common subgraph computation	Y
seq2seq	4. Plain *for* loop for computation	Y
cycleGAN	5. Miss performance tuning for static graph	Y
cholesky	6. Scalar computation that can be vectorized	Y

Table 7: Speedups by HARP-guided code refactoring.

Benchmark	No. of Ops / Edges	Library	default exec. time (s)	Speedup	Line of source codes
dict-learn	20 / 24	TensorFlow	3.09	3.0	57
rbm	83 / 102	TensorFlow	28	2.4	61
char-rnn	248 / 261	TensorFlow	25.5	1.3	462
style-transfer	129 / 131	TensorFlow	15.9	1.6	365
deep-q-net	887 / 1379	TensorFlow	58.4	2.6	534
seq2seq	999 / 1343	TensorFlow	15.9	1.5	379
cycleGAN	4721 / 5935	PyTorch	442	1.4	670
cholesky	N/A	PyTorch	0.4	2.8	109

5.2 Performance Analysis

We focus on the inefficiency patterns listed in the table 1 in the experiments. More rules can be inserted to detect more patterns. Table 6 reports the inefficiency patterns that the HARP finds in those benchmarks. It reports 1–3 potential inefficiencies in each of the benchmarks. Two out of the 12 inefficiencies are false alarms. One of the false alarms happens on `dict-learn`. HARP suggests that the Python *for* loop can be replaced with the `tf.while_loop`. Although the suggestion is valid, due to the existence of the loop redundancy, the other suggestion is more efficient. The majority of suggestions show useful insights and optimization hints. With the help of more rules, false alarms could be further reduced. It is worth noting that as these optimizations all span across both the Python host code and library APIs, none of the prior methods [18, 28, 31] can find these opportunities for their limited scopes of analysis.

Table 7 reports the speedups brought by refactoring that follows the suggestions from HARP. The baseline is the performance of the default programs running on the default TensorFlow and PyTorch systems; these systems conduct the state-of-the-art optimizations, but limit their views and optimizations to the computation graphs only. The results show 1.3-3.0X speedups, confirming that the suggestions from HARP are effective in fixing the efficiency problems that are elusive to existing optimizing systems. In the evaluation, we use the dataset included in the original application whenever it is available. We have also experimented with 2-10X larger datasets and observed similar speedups.

All these benchmarks except the `cholesky` implement some kind of machine learning algorithms as Table 5 shows. The `cholesky` measured the computations of Cholesky decomposition which is useful for efficient numerical solutions. HARP constructs an augmented computation graph for it to represent its 29 operations and host-side controls.

The `dict-learn` problem has already been discussed in Listing 2. After the enabled loop invariant elimination, its performance on GPU improves by 3.02X. For the `rmb` and `style-transfer` benchmarks, the intermediate results of multiple sub-graphs depending on the common inputs are cached and reused. The speedups are respectively 2.0X and 1.6X. For `char-rnn`, HARP suggests transforming to the `tf.while_loop` instead of the plain Python loop to implement the repeated RNN cells.

Due to the dynamic feature of the PyTorch framework, computation graphs are constructed in each iteration. This dynamic property prevents optimizations that apply to static graphs. However, for `cycGAN`, the inference by HARP leads to the conclusion that the graph is invariant across iterations. HARP hence suggests autotuning and specialization for the underlying CUDA implementation by turning on torch.backends.cudnn option. The `seq2seq` and `cholesky` contain loops that cannot be vectorized by existing methods for the unknown data types of their container objects. The analysis by HARP leads to the speculation on the uniform type of their elements and the lack of loop-carried dependencies. It suggests vectorization to the corresponding loops.

To assess the benefits brought from the speculative analysis, we disable the speculations for the several benchmarks on which speculations are used by HARP. Due to the static time unknown types, the analyzer is unable to infer the potential vectorization opportunities for `seq2seq` and `cholesky`. For another benchmark, `dict-learn`, without speculative analysis and the API knowledge, the analyzer cannot tell the loop invariant calculations.

Wrong speculations happen once in the experiments, due to an unconventional usage of decorator `@property` of Python. Codelet 9 illustrates it. Based on the first pattern in Table 3, the analyzer speculates that `batches.next` is side effect free and returns the same value in each iteration of the loop. However, the implementation of `batches.next` actually exploits the decorator `@property` and returns different values at each access. The informative feedback from HARP help programmers easily find the wrong speculations.

Table 7 also reports the numbers of nodes and edges in the computation graphs. They reflect on the complexities of the graphs, but do not necessarily indicate the complexities of the problems. For example, complex computations such as an entire CNN layer can be wrapped into a single operation in a graph.

Listing 9: Codelet causing speculation errors

```
class Batches:
    ...
    @property
    def next():
        ...
...
def train(self, session, ops, batches, n_epochs):
    for i in range(n_epochs):
        session.run(ops, feed_dict = {self.inputs:
            batches.next})
```

The analysis by HARP takes 32–973ms on these benchmarks. The refactoring based on the suggestions from HARP varies from several minutes to two hours (including debugging time), depending on the complexity of the benchmark. The benchmark `char-rnn` takes the longest time to refactor since it requires embedding dynamic control flows into static graphs. It is worth noting that as these optimizations all span across both the Python host code and library APIs, none of the prior methods [18, 28, 31] can find these opportunities for their limited scopes of analysis.

5.3 Usability Evaluation by User Study

To evaluate usability of HARP as a refactoring tool, we conduct a controlled experiment with 12 participants. The hypothesis to test is as follows: Given a sub-optimal program of interest, HARP can help users more easily find optimization opportunities and optimize the code to a more performant version.

All of the 12 participants are graduate students in computer science. Ten of them were familiar with Python programming; among the ten, three had written some beginner-level TensorFlow code before, and two had written beginner-level PyTorch code before, the other five haven't written either TensorFlow or PyTorch code before. Before the study, the participants were given a 45-minute lecture on the basics of Python, TensorFlow, and PyTorch. To eliminate the variance on experiment environments, we set up all the required software on our GPU server (Table 4) for the students to run performance measurements. The benchmarks include `dict_learn`, `char-rnn`, `rbm`) and `cycleGAN`.

We conduct a two-stage user study to examine the usability of HARP. All of the students were asked to find the inefficiencies of these programs and to optimize them if possible. In the first stage, they were not using the HARP but were allowed to use any other resources they could find. They were asked to turn in their analysis results and the optimization results in a week. Only a few students successfully optimized some of the programs (4 for `dict_learn`, 2 for `rbm`, 0 for `char-rnn` and `cycleGAN`). The speedups they got were 1.85X on `dict_learn` and 2.05X on `rbm` on the GPU server. In the second stage, the students were asked to optimize those programs with the help of HARP and submit the results in another week. The students were asked to use only the feedback provided by the HARP to optimize the programs.

Each of the programs was successfully optimized by 11 of the 12 students, and received 1.1–2.85X speedups. Program `char-rnn` is an exception. HARP suggest to use `tf.while_loop` to integrate its `for` loop into the computation graph. Students unfamiliar with TensorFlow failed making the code changes as that would need a

major rewriting of the code to embed dynamic control flows into static graphs.

In stage 1, it took the students on average 50, 88, 135, 130 mins trying to find inefficiencies in each of the four programs, and at least 72 mins to refactor a program. In Stage 2, the time in finding inefficiencies is reduced to 13–37 mins, and refactoring a program took no more than 43 mins. As stage 2 happened after stage 1, the familiarity to the code that the students had attained in stage 1 could have contributed to the reduction of the analysis and refactoring times. But the significant increase of success rates in refactoring offers evidences on the usefulness of HARP. The questionnaires accompanied with the study provided further confirmations: All students said that HARP was useful in helping them find valid optimizations. One suggestion is to make HARP offer some examples on the suggested optimizations.

5.4 Threats to Validity

Besides the possible bias in the user study mentioned earlier, this part discusses two extra threats to validity. Our framework is evaluated based on Python 3.6, TensorFlow 1.5 and PyTorch 0.4.1. As new versions are released, the APIs and the underlying implementation may change, and invalidate the refactoring patterns or optimization opportunities which are effective in the experiment. However, the goal of our work is to provide a framework to ease the analysis rather than to focus on particular patterns mentioned in this paper. New patterns are easy to be included in the future.

Although the programs in the evaluation represent real-world programs on Github, they cannot reflect the version in the process of the development. Analyzing these programs cannot simulate how our tool helps in the real programming process. HARP has the potential to give suggestions during development of a program through the integration in IDE; a detailed study is left for future.

6 RELATED WORK

There are some other work on suggesting optimizations to developers [6, 26, 27] on other programs, and many frameworks (e.g., XLA [18], TVM [4],Tensor Comprehension [36], Caffe [12]) for Deep Learning optimizations. This current study distinctively focuses on the special barrier for optimizations of Python programs that use modern machine learning libraries, and emphasizes holistic treatment.

AutoGraph [25] and JANUS [11] create Deep Learning computation graphs from imperative Python code. HARP leverages a similar idea in creating computation graphs for PyTorch code, but has a different focus, identifying inefficiencies spanning across the boundaries of host code and libraries; the inefficiencies in the evaluation section can be identified by neither AutoGraph nor JANUS.

A recent work [28] proposes a kind of intermediate representation called Weld IR to support optimizations across multiple libraries and reported promising results. It focuses on the main computations represented in the libraries rather than the API misuses or the interplay between the library calls and the host code. It requires some manual changes to the library APIs such that their calls can generate Weld IR at runtime. HARP, on the other hand, requires no modifications to the library APIs.

There is a body of work on declarative program analysis [2, 14, 37]. Speculative analysis has been explored in some prior studies on program optimizations. Lin and others [21] demonstrate that the result of speculative alias analysis can enable speculative register promotion. A recent work [34] uses several speculation techniques in the implementation of a just-in-time compiler for R.

HARP receives inspirations from the prior work. It is distinctive in being the first tool that enables holistic analysis of Python-based analytics applications, and detects large-scoped inefficiency that are elusive to existing solutions. It features novel insights on circumventing Python complexities, analytics-conscious speculative analysis, the design of *augmented computation graphs*, and the informative feedback scheme for risk controls.

7 CONCLUSION

The diverse programming paradigms and the reliance on high level abstraction are common features of modern computing. They bring challenges to the program analysis and optimizations. Current optimizers are limited in their views of the libraries, and lack the abilities to analyze across the mixed program models or to utilize high level domain knowledge. We believe that an efficient system to overcome these shortcomings needs to be able to synthesize knowledge from difference aspects of the applications of the target libraries. And a friendly interface for encoding high-level semantics of abstractions is essential. HARP is a step towards a more effective approach to advanced analysis for modern applications. It proves effective in analyzing machine learning programs, helping identify the subtle optimization opportunities for the applications. Further development directions include extending this method to a broader range of libraries and programming frameworks, and combining the approach with runtime analysis and optimization techniques.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609 and CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

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 *OSDI*, Vol. 16. 265–283.

[2] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540* (2016).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[5] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3213–3223.

[6] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2015. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *ACM*

SIGPLAN Notices, Vol. 50. ACM, 607–622.
[7] TensorFlow developers. 2016. *Implementation of Control Flow in TensorFlow*. Technical Report.
[8] Facebook and Microsoft. 2018. Open Neural Network Exchange. https://onnx.ai/.
[9] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2015. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576* (2015).
[10] Geoffrey E Hinton. 2012. A practical guide to training restricted Boltzmann machines. In *Neural networks: Tricks of the trade*. Springer, 599–619.
[11] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun. 2019. Speculative Symbolic Graph Execution of Imperative Deep Learning Programs. *ACM SIGOPS Operating Systems Review* 53, 1 (jul 2019), 26–33. https://doi.org/10.1145/3352020.3352025
[12] 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*. ACM, 675–678.
[13] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209
[14] JTransformer. 2018. The JTransformer project. http://sewiki.iai.uni-bonn.de/research/jtransformer/.
[15] Arthur Juliani. 2018. Deep Reinforcement Learning Agents. https://github.com/awjuliani/DeepRL-Agents.
[16] Andrej Karpathy. 2015. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog* (2015).
[17] Andrej Karpathy. 2016. Shakespeare Work. https://cs.stanford.edu/people/karpathy/char-rnn/shakespear.txt. Accessed: 2018-4-16.
[18] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).
[19] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010).
[20] Chen Liang. 2018. Char-RNN implemented using TensorFlow. https://github.com/crazydonkey200/tensorflow-char-rnn.
[21] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. 2003. A Compiler Framework for Speculative Analysis and Optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/781131.781164
[22] Peng Liu. 2018. RBM/DBN implementation with tensorflow. https://github.com/myme5261314/dbn_tf.
[23] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096
[24] Shahin Mahdizadehaghdam, Ashkan Panahi, Hamid Krim, and Liyi Dai. 2018. Deep Dictionary Learning: A PARametric NETwork Approach. (2018). https://doi.org/arXiv:1803.04022
[25] Dan Moldovan, James M Decker, Fei Wang, Andrew A Johnson, Brian K Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. 2018. AutoGraph: Imperative-style Coding with Graph-based Performance. (oct 2018). arXiv:1810.08061 http://arxiv.org/abs/1810.08061
[26] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 562–571.
[27] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 369–378.
[28] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.
[29] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch.
[30] Magnus Erik Hvass Pedersen. 2018. TensorFlow Tutorials. https://github.com/Hvass-Labs/TensorFlow-Tutorials.
[31] Benoıt Pradelle, Benoıt Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. 2017. Polyhedral Optimization of TensorFlow Computation Graphs. In *6th Workshop on Extreme-scale Programming Tools (ESPT-2017) at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*.
[32] Jonathan Schler, Moshe Koppel, Shlomo Argamon, and James W Pennebaker. 2006. Effects of age and gender on blogging.. In *AAAI spring symposium: Computational approaches to analyzing weblogs*, Vol. 6. 199–205.
[33] Open source. 2018. Jedi - an awesome autocompletion/static analysis library for Python. https://jedi.readthedocs.io/en/latest/
[34] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS 2016)*. ACM, New York, NY, USA, 84–95. https://doi.org/10.1145/2989225.2989236
[35] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning.. In *AAAI*, Vol. 16. 2094–2100.
[36] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730
[37] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. 2006. JunGL: A scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*. ACM, 172–181.
[38] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 97–118.
[39] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.
[40] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*.