City University of New York (CUNY)
CUNY Academic Works

Publications and Research Hunter College

2023

Towards Safe Automated Refactoring of Imperative Deep Learning Programs to Graph Execution

Raffi Takvor Khatchadourian Ph.D.
CUNY Hunter College

Tatiana Castro Vélez
CUNY Graduate Center

Mehdi Bagherzadeh
Oakland University

Nan Jia
CUNY Graduate Center

Anita Raja
CUNY Hunter College

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/hc_pubs/815
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Towards Safe Automated Refactoring of Imperative Deep Learning Programs to Graph Execution

Raffi Khatchadourian[1,2] Tatiana Castro Vélez[2] Mehdi Bagherzadeh[3] Nan Jia[2] Anita Raja[1,2]

[1]City University of New York (CUNY) Hunter College, USA

[2]City University of New York (CUNY) Graduate Center, USA

[3]Oakland University, USA

International Conference on Automated Software Engineering
September 14, 2023, Kirchberg, Luxembourg

Deep Learning Systems & Run-time Performance

- Machine Learning (ML), including Deep Learning (DL), systems are pervasive.
- As datasets grow, efficiency becomes essential to support responsiveness [Zhou et al., 2020].
- For efficiency, DL frameworks have traditionally embraced a *deferred* execution-style supporting graph-based (DNN) computation.

Scalable, but development is . . .

- Error-prone.
- Cumbersome.
- Produces programs that are difficult to debug.

- Because graph computation executes statements in a non-imperative order, traditional SE tools cannot help troubleshoot bugs [Arpteg et al., 2018].

TensorFlow Deferred Execution-style Code

```
1  # Build a graph.
2  a = tf.constant(5.0)
3  b = tf.constant(6.0)
4  c = a * b
5
6  # Launch graph in a session.
7  sess = tf.Session()
8
9  # Evaluate the tensor `c`.
10 print(sess.run(c)) # prints 30.0
```

- Lines 2–4 build a computation graph.
- Line 4 does not execute until the `Session` is run on line 10.
- No native support common imperative program constructs, e.g., iteration.

TensorFlow Deferred Execution-style Code

```
1  # Build a graph.
2  a = tf.constant(5.0)
3  b = tf.constant(6.0)
4  c = a * b
5
6  # Launch graph in a session.
7  sess = tf.Session()
8
9  # Evaluate the tensor `c`.
10 print(sess.run(c)) # prints 30.0
```

- Lines 2–4 build a computation graph.
- Line 4 does not execute until the `Session` is run on line 10.
- No native support common imperative program constructs, e.g., iteration.

TensorFlow Deferred Execution-style Code

```
1  # Build a graph.
2  a = tf.constant(5.0)
3  b = tf.constant(6.0)
4  c = a * b
5
6  # Launch graph in a session.
7  sess = tf.Session()
8
9  # Evaluate the tensor `c`.
10 print(sess.run(c)) # prints 30.0
```

- Lines 2–4 build a computation graph.
- Line 4 does not execute until the `Session` is run on line 10.
- No native support common imperative program constructs, e.g., iteration.

Imperative DL Programming, Eager Execution, & Hybridization

- Imperative DL frameworks (e.g., *TensorFlow Eager*, *Keras*, *PyTorch*) encouraging eager execution are more natural, less error-prone, and easier to debug.
- Sacrifices *run-time* performance.
- Thus, hybrid approaches (e.g., *Hybridize*, *TorchScript*, *AutoGraph*) have surfaced that:
  - Execute imperative DL programs as static *graphs* at *run-time*.
  - Are integrated into **mainstream** DL frameworks (e.g., *TensorFlow*, *MXNet*, *PyTorch*).

Eager *TensorFlow* Imperative (OO) DL Model Code

```
1 class SequentialModel(tf.keras.Model):
2   def __init__(self, **kwargs):
3     super(SequentialModel, self).__init__(...)
4     self.flatten = layers.Flatten(input_shape=(28, 28))
5     num_layers = 100 # Add many small layers.
6     self.layers = [layers.Dense(64, activation = "relu") for n in
        ↪  range(num_layers)]
7     self.dropout = tf.keras.layers.Dropout(0.2)
8     self.dense_2 = tf.keras.layers.Dense(10)
9
10
11   def __call__(self, x):
12     x = self.flatten(x)
13     for layer in self.layers:
14       x = layer(x)
15       x = self.dropout(x)
16       x = self.dense_2(x)
17     return x
```

Hybridized *TensorFlow* Imperative (OO) DL Model Code

```
 1 class SequentialModel(tf.keras.Model):
 2   def __init__(self, **kwargs):
 3     super(SequentialModel, self).__init__(...)
 4     self.flatten = layers.Flatten(input_shape=(28, 28))
 5     num_layers = 100 # Add many small layers.
 6     self.layers = [layers.Dense(64, activation = "relu") for n in
         ↪  range(num_layers)]
 7     self.dropout = tf.keras.layers.Dropout(0.2)
 8     self.dense_2 = tf.keras.layers.Dense(10)
 9
10   @tf.function(...) # Executes model as graph (optional args).
11   def __call__(self, x):
12     x = self.flatten(x)
13     for layer in self.layers:
14       x = layer(x)
15       x = self.dropout(x)
16       x = self.dense_2(x)
17     return x
```

- On line 10, *AutoGraph* used to potentially enhance performance.
- Decorates model's `call()` method with `@tf.function`, possibly providing optional yet influential decorator arguments.
- At run-time, `call()`'s execution will be "traced" (~9.22 speedup).

Hybridized *TensorFlow* Imperative (OO) DL Model Code

```
 1 class SequentialModel(tf.keras.Model):
 2   def __init__(self, **kwargs):
 3     super(SequentialModel, self).__init__(...)
 4     self.flatten = layers.Flatten(input_shape=(28, 28))
 5     num_layers = 100 # Add many small layers.
 6     self.layers = [layers.Dense(64, activation = "relu") for n in
         ↪  range(num_layers)]
 7     self.dropout = tf.keras.layers.Dropout(0.2)
 8     self.dense_2 = tf.keras.layers.Dense(10)
 9
10   @tf.function(...) # Executes model as graph (optional args).
11   def __call__(self, x):
12     x = self.flatten(x)
13     for layer in self.layers:
14       x = layer(x)
15       x = self.dropout(x)
16       x = self.dense_2(x)
17     return x
```

- On line 10, *AutoGraph* used to potentially enhance performance.
- Decorates model's `call()` method with `@tf.function`, possibly providing optional yet influential decorator arguments.
- At run-time, `call()`'s execution will be "traced" (∼9.22 speedup).

Hybridized *TensorFlow* Imperative (OO) DL Model Code

```
1  class SequentialModel(tf.keras.Model):
2    def __init__(self, **kwargs):
3      super(SequentialModel, self).__init__(...)
4      self.flatten = layers.Flatten(input_shape=(28, 28))
5      num_layers = 100 # Add many small layers.
6      self.layers = [layers.Dense(64, activation = "relu") for n in
         ↪ range(num_layers)]
7      self.dropout = tf.keras.layers.Dropout(0.2)
8      self.dense_2 = tf.keras.layers.Dense(10)
9
10   @tf.function(...) # Executes model as graph (optional args).
11   def __call__(self, x):
12     x = self.flatten(x)
13     for layer in self.layers:
14       x = layer(x)
15       x = self.dropout(x)
16       x = self.dense_2(x)
17     return x
```

- On line 10, *AutoGraph* used to potentially enhance performance.
- Decorates model's `call()` method with `@tf.function`, possibly providing optional yet influential decorator arguments.
- At run-time, `call()`'s execution will be "traced" (∼9.22 speedup).

Hybridization Drawbacks

- Needs non-trivial, specialized metadata [Jeong et al., 2019].
- Exhibit limitations and known issues with native program constructs.
- Subtle considerations required to:
  - Specify (decorate) the functions to be migrated.
  - Make code amenable to safe, accurate, and efficient graph execution.
  - Avoid performance bottlenecks and semantically inequivalent results [Cao et al., 2021, Castro Vélez et al., 2022].
- Manual analysis and refactoring (semantics-preserving, source-to-source transformation) for optimal results can be error- and omission-prone [Dig et al., 2009].
- Further complicated by:
  - Increasing Object-Orientation (OO) in DL model code [Chollet, 2020].
  - Dynamically-typed languages (e.g., Python).

Key Insight

Although imperative DL code is sequentially executed, hybridizing code resembles *parallelizing* sequential code.

Example

To void unexpected behavior, like concurrent programs, hybrid functions should avoid side-effects.

Idea

Adapt concepts from automated refactorings that parallelize sequential code, e.g., Streaming APIs [Khatchadourian et al., 2019].

Work In Progress

Two new, fully-automated refactorings are in-progress:

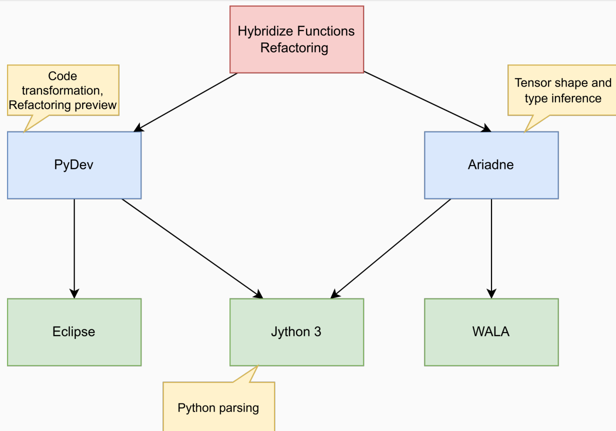
Convert Eager Function to Hybrid Transforms otherwise eagerly-executed imperative (Python) DL code for enhanced run-time performance.

- Automatically specifies (decorates) whether and how code could be reliably and efficiently executed as graphs at run-time.
- Avoids hybridizing code under certain conditions (e.g., side-effecting code) to preserve semantics.

Optimize Hybrid Function Transforms code *already* running as graphs for optimal run-time performance.

- Modifies existing decorator parameters (e.g., tensor shape specs).
- Potentially restructures code to be more amenable to graph transformation.
- Possibly *dehybridize* code when eager execution could be faster (e.g., graph "retracing").

Approach Highlights



- Novel tensor analysis for imperative DL code.
  - Current analyzers work on only procedural (TF 1) code.
  - Modernization of *WALA Ariadne* [Dolby et al., 2018] for imperative (TF 2) code underway.
- Implemented as a PyDev Eclipse IDE plug-in [Zadrozny, 2023].
  - Integrates *Ariadne* for tensor type inference and shape (static) analysis.

Approach Challenges

- Lack of static type information.
  - Needed to determine candidate functions (at least one `Tensor` parameter).
- Unlike, e.g., Java, Python has no restrictions on decorator (annotation) arguments.
- `tf.function` may be called as a function instead of a decorator.

Example

```
hyb_call = tf.function(call)
hyb_call()
```

- Determine tensor shapes.
  - Existing analyses only for *procedural* (TF 1) code.
  - Working towards statically resolving *imperative* (TF 2) code.

Conclusion

- Imperative Deep Learning code is easier to debug, write, and maintain than traditional DL code that runs in a deferred execution.
- However, it comes at the expense of (run-time) performance.
- Hybrid approaches bridge the gap between eager and graph execution.
- Using hybrid techniques to achieve optimal performance and semantics preservation is non-trivial.

Future Work

- Automated client-side analyses and transformations to *use* hybridization APIs correctly and optimally is in-progress.
- Evaluation using dataset from our MSR '22 [Castro Vélez et al., 2022] empirical study.

More details in paper! `http://bit.ly/tf2-ase23`

For Further Reading I

Abadi, Martín et al. (2016). “TensorFlow: A System for Large-Scale Machine Learning”. In: *Symposium on Operating Systems Design and Implementation.*

Agrawal, Akshay et al. (2019). *TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning*. arXiv: 1903.01855 [cs.PL].

Apache (Apr. 8, 2021). *Hybridize. Apache MXNet documentation*. URL: https://mxnet.apache.org/versions/1.8.0/api/python/docs/tutorials/packages/gluon/blocks/hybridize.html (visited on 04/08/2021).

Arpteg, A., B. Brinne, L. Crnkovic-Friis, and J. Bosch (2018). “Software Engineering Challenges of Deep Learning”. In: *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, pp. 50–59. DOI: 10.1109/SEAA.2018.00018.

Cao, Junming, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng (Dec. 3, 2021). *Characterizing Performance Bugs in Deep Learning Systems*. arXiv: 2112.01771 [cs.SE].

Castro Vélez, Tatiana, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja (May 2022). “Challenges in Migrating Imperative Deep Learning Programs to Graph Execution: An Empirical Study”. In: *International Conference on Mining Software Repositories*. MSR '22. ACM/IEEE. ACM. DOI: 10.1145/3524842.3528455. arXiv: 2201.09953 [cs.SE].

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang (2015). “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. In: *Workshop on Machine Learning Systems at NIPS*. arXiv: 1512.01274 [cs.DC].

Chollet, François (2020). *Deep Learning with Python*. 2nd ed. Manning.

Dig, Danny, John Marrero, and Michael D. Ernst (2009). “Refactoring sequential Java code for concurrency via concurrent libraries”. In: *International Conference on Software Engineering*. IEEE, pp. 397–407. DOI: 10.1109/ICSE.2009.5070539.

Dilhara, Malinda, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig (2022). “Discovering Repetitive Code Changes in Python ML Systems”. In: *International Conference on Software Engineering*. ICSE '22. To appear.

For Further Reading II

Dolby, Julian, Avraham Shinnar, Allison Allain, and Jenna Reinen (2018). “Ariadne: Analysis for Machine Learning Programs”. In: *International Workshop on Machine Learning and Programming Languages*. MAPL 2018. ACM SIGPLAN. Philadelphia, PA, USA: Association for Computing Machinery, pp. 1–10. ISBN: 9781450358347. DOI: 10.1145/3211346.3211349.

Facebook Inc. (2019). *PyTorch Documentation. TorchScript*. en. URL: https://pytorch.org/docs/stable/jit.html (visited on 02/19/2021).

Jeong, Eunji, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun (July 2019). “Speculative Symbolic Graph Execution of Imperative Deep Learning Programs”. In: *SIGOPS Oper. Syst. Rev.* 53.1, pp. 26–33. ISSN: 0163-5980. DOI: 10.1145/3352020.3352025.

Khatchadourian, Raffi, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed (2019). “Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams”. In: *International Conference on Software Engineering*. ICSE ’19. IEEE Press, pp. 619–630. DOI: 10.1109/ICSE.2019.00072.

Kim, Miryung, Thomas Zimmermann, and Nachiappan Nagappan (Nov. 2012). “A Field Study of Refactoring Challenges and Benefits”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina: ACM. ISBN: 9781450316149. DOI: 10.1145/2393596.2393655.

Moldovan, Dan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko (2019). *AutoGraph: Imperative-style Coding with Graph-based Performance*. arXiv: 1810.08061 [cs.PL].

Negara, Stas, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig (2013). “A Comparative Study of Manual and Automated Refactorings”. In: *European Conference on Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 552–576. ISBN: 978-3-642-39038-8.

OpenAI, Inc. (Aug. 18, 2023). *ChatGPT*. URL: https://chat.openai.com (visited on 08/18/2023).

Paszke, Adam et al. (Dec. 3, 2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. arXiv: 1912.01703 [cs.LG].

For Further Reading III

Zadrozny, Fabio (Apr. 15, 2023). *PyDev*. URL: https://www.pydev.org (visited on 05/31/2023).

Zhou, Weijie, Yue Zhao, Guoqiang Zhang, and Xipeng Shen (2020). "HARP: Holistic Analysis for Refactoring Python-Based Analytics Programs". In: *International Conference on Software Engineering*, pp. 506–517. DOI: 10.1145/3377811.3380434.

Appendix

Why Static Analysis?

- Refactorings must operate on (at least some) static information.
- Must eventually transform the *source* code.
- May eventually integrate hybrid analyses to resolve difficult static cases.

Why Automated Refactoring?

- In general, such problems may also be handled by compilers or runtimes; however, refactoring has several benefits:
  - Gives developers more control over where the optimizations take place and making graph execution explicit.
  - Can be issued multiple times, e.g., prior to major releases.
  - Unlike static checkers, they transform source code, a task that can be otherwise error-prone and involve subtle nuances.
- Refactorings can act like recommendation systems, which is important for analyzing and transforming programs written in dynamic languages where static assumptions may be easily violated!

Refactoring Developer Adoption

- Developers generally underuse automated refactorings [Kim et al., 2012, Negara et al., 2013].
- Data scientists and engineers may be more open to using automated (refactoring) tools.
- Our approach will be fully automated with minimal barrier to entry.

LLMs & Big Data Refactoring

- LLMs [OpenAI, Inc., 2023] can also perform refactorings.
- Other Big Data-driven refactorings [Dilhara et al., 2022] are exciting and promising.
- Obtaining a (correct) dataset large enough to automatically extract the proposed refactorings is challenging as developers struggle with (manually) migrating DL code to graph execution [Castro Vélez et al., 2022].
- LLM inference capabilities are currently limited.
  - LLMs have a token limitation.
  - Hybridization requires *interprocedural* analysis.

Notebook Support

- We plan to investigate notebook support in the future.
- We envision the approach to be used on (larger) DL *systems*, consisting of multiple files.