An Empirical Study on the Evolution of Deep Learning Library Usages

Benjamin J. Rombaut[1]
[1] *School of Computing, Queen's University, Canada*

***Abstract*—Deep Learning techniques have become prevalent in various domains, and a growing number of open source projects in GitHub rely on deep learning libraries to implement their algorithms. These libraries have undergone substantial changes since their inception, with fast releases consisting of new features, bug fixes, and deprecating out of date APIs. In this work, we study how the usages of four commonly used deep learning libraries, namely TensorFlow, PyTorch, Keras and Theano, evolve over time. We find that, while Tensorflow is the most popular library, PyTorch is rapidly gaining traction, and Theano is seeing its usage decline. These libraries are usually imported with other base Python packages, as well as specialized deep learning packages from the same vendor. Analysis on the evolution of call statements from deep learning libraries is still ongoing.**

Source-code Link: https://github.com/brombaut/CISC873_TermProject

***Keywords*-Deep Learning, API Usages, Framework Evolution**

I. Introduction

An emerging branch of machine learning algorithms known as deep learning algorithms has attracted considerable attentions in both academia and industry recently [1]. Due to the increasing popularity of deep learning, there are many empirical studies in software engineering domains that look into deep learning code. Examples include detecting and locating code mistakes in deep learning applications [2], understanding the bug types, common anti-patterns, and good and bad practices in deep learning projects [3] [4], characterizing deep learning development and deployment across different frameworks and platforms [1], and the applications domains and update behaviours of client projects of deep learning libraries [5]. Among them, very few studies have quantitatively analyzed the evolution of deep learning library usages, how client's are actually making use of their APIs, what common libraries are used with these deep learning APIs, or how these practices evolve over time.

Deep learning libraries are constantly evolving to add new features and fix bugs. As new versions of these libraries become available, how client's make use of these libraries evolves as well. Additionally, client's may make use of other common libraries or APIs in tandem with these deep deep learning APIs. Therefore, an exploratory study to understand the evolution of deep learning library uses can shed light on the development and improvement of deep learning libraries, and provide practical suggestions to developers, users, and researchers.

To achieve this goal, in this paper, we analyze the popularity and usage over time of four typical and widely studies deep learning libraries: TensorFlow[1], PyTorch[2], Theano[3] and Keras[4] [6] [7] [8]. We examine what other libraries tend to be used with these libraries, as well which library function calls are used most by clients, and how these usages evolve over time. Specifically, we investigate the following research questions:

- *RQ1: What deep learning libraries do clients use?*
- *RQ2: What libraries are commonly imported alongside deep learning libraries?*
- *RQ3: What deep learning library calls are used together?*

The remainder of this paper is structured as follows. We start by providing an overview of the study design in Section II. Section III discusses the data set we used for the study. Section IV presents the results of our research questions. Section V presents the threats to validity of our study. The related work is presented in Section VI. Finally, Section VII draws conclusions and discusses future work.

II. Approach

To study the evolution of deep learning library usages, we collected the open source projects that depend on TensorFlow, PyTorch, Theano and Keras in GitHub. In this section, we discuss our methodology for how we would answer the research questions of our study. The collected data will be analyzed in Section III.

A straw-man technique to determine the libraries that a project is using would be to read the projects *requirements.txt*[5] file to extract the dependencies and versions required by the project. However, it is not uncommon for projects to fail specify the versions of libraries they depend on, or to even disregard including a *requirements.txt* file at all in their projects. Pan et al. [5] found in their study of dependency networks in deep learning libraries that at least one-fifth of the deep learning projects include no requirement files or no version statements in requirement files, and that their results indicate that there are too many deep learning projects that have no version management of deep learning libraries, which is detrimental to themselves and those projects that depend on them, and are prone to yield crashes or security issues. In terms of our study, this would mean we would not be able to gather information on which deep learning libraries these project are using.

[1] https://www.tensorflow.org/
[2] https://pytorch.org/
[3] http://deeplearning.net/software/theano/
[4] https://keras.io/
[5] https://pip.pypa.io/en/stable/user_guide/#requirements-files

To determine what libraries a project is using, we decide to perform static analysis on the source code to directly extract the libraries the project is importing in each file. This method will provide a much more reliable and accurate evaluation of which libraries the project depends on, and give a clear picture of how these dependencies evolve over the lifetime of the project.

To determine how deep learning libraries are used by clients, we extract the call statements from the python source files that import deep learning libraries. This includes all call statements made using the deep learning library APIs, as well as any other call made from the client in the file being analyzed.

An additional attribute we will need to answer our research questions is the parent function each library function is called from in the client's project. These attributes will allow us to determine what functions from deep learning libraries are most commonly used overall, as well as which are most commonly used together.

To examine how the usages of deep learning libraries evolve over time, we calculate the differences between the imports and call statements for each file across all the releases of a project.

III. Data Set

In this section, we discuss how we collected the data set used in this study.

A. Project Selection

We construct our initial list of Deep Learning project to examine from four different sources. We begin with the list of projects that were examined by Jebnoun et al. in their study of code smells in Deep Learning projects [4]. We also collect the most popular projects from the *Awesome Open Source*[6] database. We select the first 100 projects ordered by the number of GitHub stars from the *self driving car*[7] category, the *object detection*[8] category, and the *artificial intelligence*[9] category. We try to avoid biasing our results from our project selection by examining projects from different domains and general categories, since it is not unlikely that deep learning projects from the same specific domain would use similar deep learning libraries. In total, our list consists of 309 unique projects.

B. Releases

Since we look to examine how the use of deep learning libraries evolve over time, we need to collect snapshots of each project throughout their lifetime. To do this, we retrieve the list of tags that a projects has. Typically, these tags are used by maintainers to track releases of their projects[10]. If the project does not have any tags associated with it, we do not consider it in our analysis. In total, we find that 176 of the projects we collected in Section III-A have at least 1 release, for a total of 3629 releases during the time period between January 24, 2004 and December 12, 2020.

C. Import Statements

We clone each of the deep learning projects that have at least 1 release, and check-out each release in chronological order. As previously mentioned, simply analyzing the *requirements.txt* file to determine what libraries the project is using proved to be very ineffective, we decide to do static analysis on the source code directly. For every Python file contained in the project folder, we create the Abstract Syntax Tree (AST) from it's source code using the built-in python AST package[11]. We then walk the parsed AST of the source file, looking for any import nodes that import either the Tensorflow, PyTorch, Theano or Keras libraries.

In Python, import statements can be written in multiple ways. Figure 1 shows an example of three of the most common types of imports. On line 1, the *keras* library is being imported directly using a base import statement. On line 2, the *tensorflow* library is being imported directly, but will be referenced in the code using the alias *tf*. Finally, on line 3, the *nn* resource is being imported from the *torch* library. Table I shows how each of these import statements would be parsed.

```
1  import keras
2  import tensorflow as tf
3  from torch import nn
```

Figure 1: Example of different types of import statements in Python

Statement	Import	Alias	Module	Library
import keras	keras	Null	Null	keras
import tensorflow as tf	tensorflow	tf	Null	tensorflow
from torch import nn	nn	Null	torch	torch

Table I: How each type of import is parsed

If we find that the AST of a source file in the project is importing a deep learning library, we mark this file as a candidate to be analyzed. As we are interested in the evolution of how client's use deep learning libraries, rather than analyzing every source file in the project, we only analyze files that make use of the deep learning libraries. We then collect all the import statements in these files.

In total, we collect 691,232 import statements.

D. Call Statements

Using the same list of python files we determined to be importing at least one of the deep learning libraries from Section III-C, we also collect the call statements from the file. We again walk the AST of the python source code to find and extract the information of all the call nodes in the tree.

[6]https://awesomeopensource.com/

[7]https://awesomeopensource.com/projects/self-driving-car

[8]https://awesomeopensource.com/projects/object-detection

[9]https://awesomeopensource.com/projects/artificial-intelligence

[10]https://git-scm.com/book/en/v2/Git-Basics-Tagging

[11]https://docs.python.org/3/library/ast.html

We collect the full name of the function call, including the module name if the function is called from a module chain, as well as the parameter names that are passed to the function call. In order to determine what methods are called from the same section of code, we also collect the parent function that each call statement is called from.

An example of a code snipped with function calls can be seen in Figure 2, and the associated data that we would have extracted from this snipped is given in Table II.

In total, we collect 5,792,864 call statements.

```
1  def tensor_shapes_match(a, b):
2      return tf.shape(a).shape == tf.shape(b).shape
3
4  tensor_shapes_match(m_a, m_b)
```

Figure 2: Code snipped with call statements

Call name	Parameters	Parent Function
tf.shape	{0: 'a'}	tensor_shapes_match
tf.shape	{0: 'b'}	tensor_shapes_match
tensor_shapes_match	{0: 'm_a', 1: 'm_b'}	Null

Table II: Call statements parsed from Figure 2

E. Evolution across releases

Once we had collected all the import and call statements across all releases of the projects, we parsed the differences between these two entities across releases sorted chronologically.

In order to parse the import diffs across releases for a project, we track the import signatures that each file in the project has at each specific release. We then determine how the import signatures change for each file across each release. For example, if a file at Release A contains an import statement that is not present in the file at the next release, Release B, we consider that import statement to have been added between Release A and Release B. If a file at Release B contains an import statement that is not present in the file at the previous release, Release A, then we consider that import statement to have been added between Release A and Release B.

We follow the same process of parsing the call diffs across releases. If a file at Release A contains a call signature that is not present in the file at the next release, Release B, we consider that call signature to have been deleted between Release A and Release B. If a file at Release B contains a call signature that is not present in the file at the previous release, Release A, then we consider that call signature to have been added between Release A and Release B.

In addition to parsing the import and call diffs at the file level across releases, we also generalize the approach and parse the import and call diffs at the project level. For example, if Release A of a project contains at least one file that contains a specific import statement, but the next release, Release B, of the project does not contain any files that contain the same import statement, then we consider that imported library to have been deleted from the project between Release A and Release B. If Release B of a project contains at least one file that contains a specific import statement, but the previous release, Release A, did not have any files that contain that import statement, then we consider the imported library to have been added to the project between Release A and Release B. The same process is used for call statements.

Table III summarizes the total data set that was collected/

Entity	Records
Projects	309
Projects ($\geq$ 2 Releases)	176
Releases	3,629
Imports (Project Level)	205,196
Imports (File Level)	691,232
Calls (Project Level)	1,478,643
Calls (File Level)	5,792,864

Table III: Summary of the data set

IV. Results

In this section, we present the results of our empirical study with respect to our three research questions. For each research question, we present our motivation, approach, and results.

A. RQ1: What deep learning libraries do clients use?

1) Motivation: The first step we wanted to take when examining the evolution of deep learning library usages was to see which of the deep learning libraries client's were using the most in their projects, and if these trends evolve over time. This will provide a good first indicator of whether certain deep learning libraries are gaining or losing popularity amongst clients.

2) Approach: To determine which deep learning libraries client's were using in their projects, we examine all the libraries clients are importing in their source files, generalize this information to the project level, and track the differences across each release of each project. As previously mentioned in Section III-C, this is a much more reliable process for determining the libraries a python project is using rather than looking at the *requirements.txt* file. We track the number of client projects that are importing each deep learning library. Additionally, since the projects we examine may have been released at different times, we also track the proportion of projects that are importing each deep learning library. Figure 3 shows the proportion of projects we examined that have had their first release.

3) Results: Figure 4 shows the total number of projects from our data set that import each of the deep learning libraries, as well as the proportion of projects that have had their first release that are importing each deep learning library. We note that about 2/3 of projects have been released by 2019.

We found that TensorFlow is the most widely used deep learning library amongst clients. After 2018, the Tensorflow library is imported into roughly 40% of the projects we

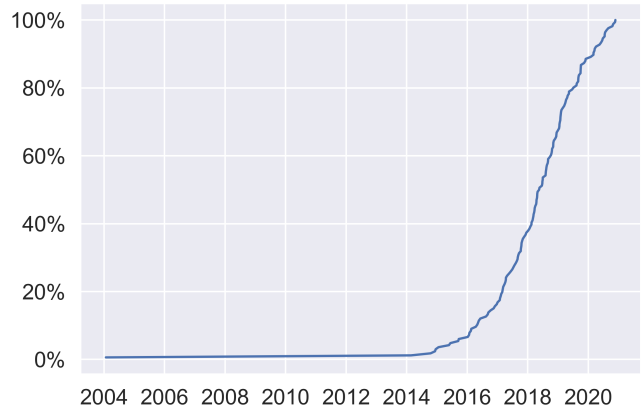

Figure 3: Proportion of examined projects that have been released

examined. PyTorch has been steadily rising in popularity since its initial release in September 2016, and is now used in over 30% of projects. Keras is used in over 10% of clients, while Theano's is used in under 10% of clients, and has seen it's popularity steadily decreasing.

To summarize, our results indicate that, while TensorFlow is still the most popular deep learning library, PyTorch is rapidly gaining traction. Both of these deep learning libraries have a wide client base and strong support in the ecosystem. The decreasing trend in popularity for Theano reflects that fact that maintenance of the library was ceased in September 2017.

> Approximately 40% and 10% of projects use the TensorFlow library and the Keras library, respectively. The popularity of PyTorch has been rising steadily over the previous years, and is now used by approximately 30% of clients. Theano is the least popular library.

B. RQ2: What libraries are commonly imported alongside deep learning libraries?

1) Motivation: Clients that use deep learning libraries often use additional libraries in their projects. Maintainers of deep learning libraries can learn what other packages are being used in tandem with their, as well as what specific portions of theor libraries are being used the most.

2) Approach: To determine what other packages evolve with deep learning libraries, we examine the import statements of each file that import at least one of the deep learning libraries (i.e. TensorFlow, PyTorch, Theano, or Keras). We use the apriori [9] algorithm implemented in the apyori[12] package to extract frequent item-sets from the following data sets:

- libraries that are imported in the same file together
- libraries that are added to the same file together
- libraries that are removed from the same file together

We then select item-sets contain at least one of the deep learning libraries (i.e. TensorFlow, PyTorch, Theano, or Keras) to determine which other libraries are evolving with the deep learning libraries.

3) Results: Table IV highlights the five item-sets with the highest lift from the previously mentioned categories. We did not have enough data to generate any item-sets for the Theano library, so it is omitted from the table.

Base python modules, such as *os*[13], *argparse*[14], and *sys*[15], are commonly imported with all deep learning libraries. Additionally, the *future* package is commonly imported, which is described as "the missing compatibility layer between Python 2 and Python 3."[16]. The *NumPy*[17] package is also often imported in tandem with deep learning libraries.

We found that the *tensorpack* [10] and *tensorlayer* [11] libraries tend to be used and evolve with TensorFlow. Both of these packages are TensorFlow-based deep learning and reinforcemembt learning libraries. Files that import the PyTorch module also import *torch.nn*[18] and other API modules from the PyTorch family. Files that import the Keras module on it's own usually also import additional libraries from the Keras module. For example, it is common for files that import *keras* to also import *keras.models*[19] and *keras.layers*[20]. We were only able to extract a single frequent item-set in regards to what libraries are added or removed with *keras*, which was *keras.layers*. The full statistics of the common item-sets are available in Table IV.

> Deep learning libraries tend to be imported with common base python modules, such as *os*, *argparse*, and *sys*. Other, more specialised sub-modules of the same deep learning library tend to be imported alongside the base deep learning library itself.

C. RQ3: What deep learning library calls are used together?

1) Motivation: Examining the evolution of how client's are using specific calls from the deep learning APIs swill allow maintainers of deep learning libraries to gain insight into how their clients are reacting to changes in their API. This will also provide a starting point to look at the cost incurred by clients when changing the deep learning APIs they use, as well as how long it takes for client's to react to deprecated APIs in the deep learning libraries.

2) Approach: To determine what calls are being used alongside calls from deep learning library APIs, we examine the call statements from each project, grouped by their parent function. We then use these grouped calls as input records to the apriori algorithm described in the previous research question to extract frequent item-sets from the following data sets:

[12]https://pypi.org/project/apyori/

[13]https://docs.python.org/3/library/os.html
[14]https://docs.python.org/3/library/argparse.html
[15]https://docs.python.org/3/library/sys.html
[16]https://pypi.org/project/future/
[17]https://numpy.org/
[18]https://pytorch.org/docs/stable/nn.html
[19]https://keras.io/api/models/
[20]https://keras.io/api/layers/

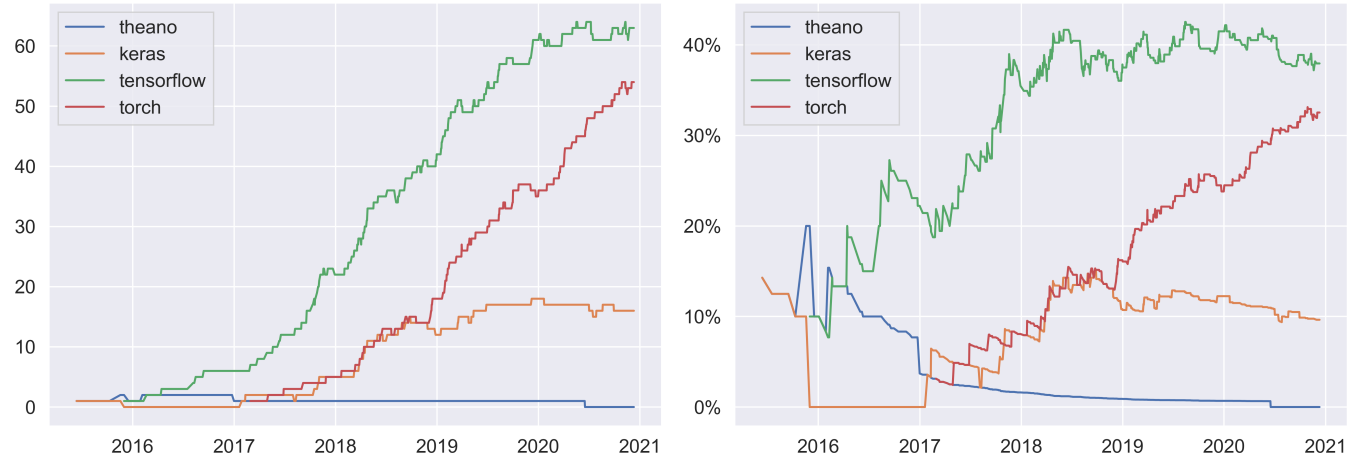

Figure 4: Number of projects importing deep learning libraries (left) and proportion of active projects importing deep learning libraries (right)

- calls that are added with the same parent function together across releases
- calls that are removed with the same parent function together across releases

3) Results: While work for this research question is still ongoing, Table V highlights the five item-sets with the highest lift from the previously mentioned categories. We could not extract any meaningful conclusions from this data set, and feel that our methodology will have to be reworked in the future to effectively answer this research question.

V. Threats to Validity

In this section, we disuses threats to validity that might influence our study.

A. Threats to Internal Validity

Internal validity concerns factors that could have influenced our analysis and findings. One threat can be attributed to the trustworthiness of our sample projects. While we specifically selected the projects with the most GitHub stars from the *awesome open source* data sets and used projects selected from a different study on deep learning dependency networks [5], these projects may not be dispersed across different enough domains for our results to be able to generalize to most deep learning libraries. We also filter out nearly half of our projects because they are not tagged with multiple releases, which again may have biased our project selection.

As previously mentioned, we use the Python AST package to parse the Python source code and extract the import and call statements. Another thread can be attributed to whether the projects we analyze are using different versions of Python, since the AST module uses the same parser to compile the Python source code into byte code as is on the machine that it is running. This means that we would have been using the parser for Pytnon 3. While import and call statements remain relatively unchanged between versions, we may have not been able to correctly parse projects that use version 2 of Python if the parser received tokens it did not recognize.

B. Threats to External Validity

Threats to external validity concern the generalization of our technique and findings. While we studied a sample of open source projects that depend on Tensorflow, PyTorch, Theano and Keras, results of this paper may not extend to all deep learning libraries. However, due to that the four deep learning libraries are all typical and popular, we believe that our findings can provide insight to practitioners that use other deep learning libraries concerning how the uses of deep learning libraries in general are used. Additionally, since we only looked at open source projects, our results may not generalize to proprietary deep learning systems.

VI. Related Work

In this section, we present the work most related to our study.

Pan et al. [5] conducted a study in 2020 on dependency networks of deep learning libraries. They examined the domains of projects that depend on deep learning libraries, to what extend these projects depend on deep learning libraries, and what the update behaviors are of client projects that depend on deep learning libraries. They found that Tensorflow-dependent and PyTorch-dependent projects that provide replication packages of research papers have more contributors and stars, the image/video processing, NLP, model theory, and efficiency library applications are the most common application domains, projects account for a higher proportion of direct dependencies than transitive dependencies on deep learning libraries, and only a small percentage of projects have upgraded deep learning libraries out of various reasons. We look to build on their work by examining more specifically how a client's usage of these libraries evolve over time.

Wu et al. [12] performed an exploratory study in 2016 on api changes and usages in the apacge and eclipse ecosystem. They

Frequency Type	Deep Learning Library	Item-set	Support	Confidence	Lift
Imported Together	TensorFlow	(tensorpack, tensorpack.tfutils.summary)	0.010252	0.909677	41.543110
		(os, tensorpack, argparse, numpy)	0.011451	0.522966	29.418010
		(argparse, tensorpack, os)	0.015244	0.696182	29.133008
		(tensorlayer, numpy, tensorlayer.layers)	0.011585	0.789430	28.472623
		(tensorlayer, time, tensorlayer.layers)	0.010797	0.735756	28.135320
	PyTorch	(pytest, pytorch_lightning, tests.base)	0.011451	0.567227	33.434784
		(numpy.testing, allennlp.common.testing)	0.011161	0.785166	20.628299
		(pytest, allennlp.common.testing, allennlp.common.checks)	0.011706	0.939689	16.636986
		(allennlp.data, allennlp.common.checks)	0.011161	0.599219	15.608829
		(pytorch_lightning, tests.base)	0.012421	0.615246	15.155623
	Keras	(keras.models, keras.layers)	0.021000	0.536035	20.914729
		(keras.models, keras.layers, numpy)	0.016359	0.664697	19.185779
		(keras.models, keras.layers, __future__)	0.010640	0.592043	17.430105
		(keras.models, __future__)	0.013342	0.742414	16.630155
		(keras.layers, __future__)	0.011500	0.639919	16.333870
Added Together	TensorFlow	(tensorlayer, time)	0.011615	0.537162	12.743870
		(sonnet.src, __future__)	0.012711	1.000000	10.185268
		(gpflow, numpy)	0.010739	0.807692	6.905996
		(os, sys)	0.012200	0.784038	5.478658
		(argparse, os)	0.011177	0.739130	5.164858
	PyTorch	(torch.nn, torch.nn.functional)	0.021185	0.625000	11.546053
		(argparse, os)	0.013734	0.604502	4.224105
		(os, torch.nn)	0.011688	0.946746	4.060150
		(torch.nn, torch.utils.data)	0.011104	0.932515	3.999124
		(torch.nn.functional)	0.030755	0.907328	3.891105
	Keras	(keras.layers)	0.011615	0.517915	18.132334
Removed Together	TensorFlow	(tensorlayer, time, numpy)	0.010843	0.833333	30.469897
		(tensorlayer, time)	0.013976	0.517857	18.934865
		(sonnet.src, __future__)	0.010241	1.000000	8.773784
		(absl.testing, __future__)	0.011205	0.823009	7.220902
		(gpflow, numpy)	0.012410	0.895652	7.039690
	PyTorch	(torch.nn, torch.nn.functional)	0.022169	0.607261	11.175752
		(os, pytorch_lightning)	0.016145	0.519380	10.239555
		(torch.nn, torch.utils.data)	0.011687	0.544944	8.534026
		(argparse, os)	0.013855	0.761589	5.579163
		(os, sys)	0.010120	0.688525	5.043914

Table IV: Five most common import item-sets of deep learning libraries for imports that appear in the same file, that are added together, and that are remove together (support $\geq$ 0.01, confidence $\geq$ 0.5, lift $\geq$ 3)

found that missing classes and methods happen more often in frameworks and affect client programs more often than the other API change types do, missing interfaces occur rarely in frameworks but affect client programs often, framework APIs are used on average in 35% of client classes and interfaces, most of such usages could be encapsulated locally and reduced in number, and about 11% of APIs usages could cause ripple effects in client programs when these APIs change. While their results do not transfer directly to deep learning ecosystem or the python language in general, their work is still highly relevant due to the similarities in examining api usages.

Wittern et al. [13] studied the evolution of the npm JavaScript library ecosystem and analyzed their characteristics such as dependencies, popularity, version distribution, etc. The insights they found in their study can help understand the evolution of npm, design better package recommendation engines, and can help developers understand how their packages are being used. Our study looks to accomplish the same goal for deep learning library ecosystem.

Although there have been many studies on the evolution of dependency networks across various programming language ecosystems, there has not yet been any studies that look at how deep learning libraries are used by clients. Our study can shed light on how client's usages of deep learning libraries evolves over time and provide new research directions such as how client's react to deprecation of deep learning APIs and what is the cost incurred by clients when having to change their API usage.

VII. Conclusion

In this paper, we lay the groundwork for studying the evolution of deep learning library usages. We analyze 309 open-source projects all make use of common deep learning libraries, namely Tensorflow, PyTorchm, Keras, or Theano. We extract the imports each project makes in the source files that

Frequency Type	Deep Learning Library	Item-set	Support	Confidence	Lift
Added Together	TensorFlow	(tape.gradient, tf.GradientTape)	0.001142	0.902439	695.952381
		(tf.Graph(), tf.Graph)	0.001683	0.542289	305.473502
		(tf.Session, tf.ConfigProto)	0.001343	0.769912	160.886669
		(self.test_session, sess.run, tf.global_variables_initializer)	0.001652	0.963964	116.069862
		(tf.global_variables_initializer, sess.run, tf.Session)	0.001003	0.631068	113.242611
	PyTorch	(torch.no_grad, model.eval)	0.001173	0.513514	118.805019
		(super, super, torch.nn.Linear)	0.001173	0.500000	22.508687
		(super, torch.nn.Linear)	0.001173	0.500000	20.500000
		(super, torch.nn.Linear)	0.001575	0.671053	15.530829
		(len, torch.argmax)	0.001065	0.696970	14.144642
Removed Together	TensorFlow	(saver.save, tf.train.Saver)	0.001135	0.758065	261.557527
		(tf.Graph(), tf.Graph)	0.001860	0.962500	249.070938
		(tf.constant_initializer, tf.truncated_normal_initializer)	0.001256	0.597701	156.627964
		(tf.argmax, tf.reduce_mean, tf.equal)	0.001208	0.862069	128.855969
		(tf.Session, tf.ConfigProto)	0.001522	0.741176	116.241176
	PyTorch	(len, torch.sum)	0.001111	0.901961	778.016340
		(torch.sum, torch.tensor)	0.001014	0.823529	668.576701
		(torch.sum, torch.argmax, torch.tensor)	0.001014	0.608696	600.057971
		(len, torch.argmax, torch.sum)	0.001111	0.666667	600.057971

Table V: Five most common call item-sets of deep learning libraries for calls that are added and remove together (support $\geq$ 0.001, confidence $\geq$ 0.5, lift $\geq$ 3)

use the deep learning libraries, as well as the call statements, and determine frequent item-sets of imports and calls that evolve together.

For future work, it would be beneficial perform our analysis on more diverse set of project to expand the generalization of our results. While we are confident our methodology for evaluating the evolution of packages being imported alongside deep learning libraries, we are less assure about our approach for evaluating the evolution of calls being made both from the deep learning library, as well as from other package dependencies.

REFERENCES

[1] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," *CoRR*, vol. abs/1909.06727, 2019. [Online]. Available: http://arxiv.org/abs/1909.06727

[2] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 826–837. [Online]. Available: https://doi.org/10.1145/3368089.3409720

[3] M. Xiu, E. E. Eghan, Z. Ming, Jiang, and B. Adams, "Empirical study on the software engineering practices in open source ml package repositories," 2020.

[4] H. Jebnoun, H. B. Braiek, M. M. Rahman, and F. Khomh, "The scent of deep learning code: An empirical study," *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

[5] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, and X. Xia, "An empirical study of the dependency networks of deep learning libraries," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 868–878.

[6] J. Han, E. Shihab, Z. Wan, S. Deng, and X. Xia, "What do programmers discuss about deep learning frameworks," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2694–2747, Jul 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09819-6

[7] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," *CoRR*, vol. abs/1906.01388, 2019. [Online]. Available: http://arxiv.org/abs/1906.01388

[8] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[9] H. Toivonen, *Apriori Algorithm*. Boston, MA: Springer US, 2010, pp. 39–40. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_27

[10] Y. Wu *et al.*, "Tensorpack," https://github.com/tensorpack/, 2016.

[11] H. Dong, A. Supratak, L. Mai, F. Liu, A. Oehmichen, S. Yu, and Y. Guo, "TensorLayer: A Versatile Library for Efficient Deep Learning Development," *ACM Multimedia*, 2017. [Online]. Available: http://tensorlayer.org

[12] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, Dec 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9411-7

[13] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 351–361. [Online]. Available: https://doi.org/10.1145/2901739.2901743