An Exploratory Study of Deep Learning Supply Chain

Xin Tan
School of Computer Science and Engineering
Beihang University
State Key Laboratory of Software Development Environment
Beijing, China
xintan@buaa.edu.cn

Kai Gao
School of Software & Microelectronics
Peking University
Beijing, China
gaokai19@pku.edu.cn

Minghui Zhou*
School of Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, China
zhmh@pku.edu.cn

Li Zhang
School of Computer Science and Engineering
Beihang University
State Key Laboratory of Software Development Environment
Beijing, China
lily@buaa.edu.cn

ABSTRACT

Deep learning becomes the driving force behind many contemporary technologies and has been successfully applied in many fields. Through software dependencies, a multi-layer supply chain (SC) with a deep learning framework as the core and substantial downstream projects as the periphery has gradually formed and is constantly developing. However, basic knowledge about the structure and characteristics of the SC is lacking, which hinders effective support for its sustainable development. Previous studies on software SC usually focus on the packages in different registries without paying attention to the SCs derived from a single project. We present an empirical study on two deep learning SCs: TensorFlow and PyTorch SCs. By constructing and analyzing their SCs, we aim to understand their structure, application domains, and evolutionary factors. We find that both SCs exhibit a short and sparse hierarchy structure. Overall, the relative growth of new projects increases month by month. Projects have a tendency to attract downstream projects shortly after the release of their packages, later the growth becomes faster and tends to stabilize. We propose three criteria to identify vulnerabilities and identify 51 types of packages and 26 types of projects involved in the two SCs. A comparison reveals their similarities and differences, e.g., TensorFlow SC provides a wealth of packages in experiment result analysis, while PyTorch SC contains more specific framework packages. By fitting the GAM model, we find that the number of dependent packages is significantly negatively associated with the number of downstream projects, but the relationship with the number of authors is nonlinear. Our findings can help further open the "black box" of deep learning SCs and provide insights for their healthy and sustainable development.

*Corresponding author

CCS CONCEPTS

• **Software and its engineering → Software evolution**; *Software libraries and repositories*; *Risk management*.

KEYWORDS

software supply chain, deep learning, open source, structure, evolution

ACM Reference Format:
Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. 2022. An Exploratory Study of Deep Learning Supply Chain. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510199

1 INTRODUCTION

In recent years, deep learning (DL) has gradually become the research hotspot and mainstream direction in the field of artificial intelligence [26]. DL allows computational models of multiple processing layers to learn and represent data mimicking how the brain perceives and understands multimodal information, thus implicitly capturing intricate structures of large-scale data [41]. Due to its high performance, it has been applied in various fields, e.g., computer vision (CV) and natural language processing (NLP), and used by thousands and millions of people every day [16].

Behind the boom of DL techniques, DL frameworks are springing up in universities and companies, among which some excellent DL frameworks also emerged, such as TensorFlow[1] and PyTorch[2]. Based on these frameworks, developers design their own applications to handle various tasks and even release relevant software packages to enrich and support the existing DL ecosystems [13]. In addition to providing different functions, these DL packages also act as bridge nodes attracting their own downstream projects and

[1]https://www.tensorflow.org
[2]https://pytorch.org

gradually form a supply chain (SC) with a DL framework as the core and substantial downstream projects as the periphery. The SCs derived from dependent relationships can dynamically reflect the overall picture of the ecosystem formed from the DL framework.

However, the DL SCs remain "black boxes" due to the large number of projects and intricate dependencies involved in the SCs, as well as a lack of effective modeling methods [18, 46]. It makes many basic problems unclear, e.g., structure, which hinders the effective support for its long-term development. For example, some DL enthusiasts noticed that many researchers are shifting from TensorFlow to PyTorch, which is a bad signal for the TensorFlow community [23]. To formulate effective countermeasures, it is necessary to quantify the development state of the two DL SCs and compare their differences. Worse still, several previously prominent DL frameworks gradually have been deprecated, e.g., caffe2 and CNTK, so the issue of how to build the ecosystem of an emerging framework requires urgent solutions [7, 43]. Given the great success of DL frameworks such as TensorFlow, investigation on their SCs can provide references for the development of these emerging frameworks. Additionally, previous studies of SCs usually only focus on the packages in different registries, e.g., Npm [15, 17, 44], which makes the evolution and related factors of sophisticated SCs derived from a single package as a knowledge gap.

To bridge this knowledge gap and promote the long-term development of DL SCs, this study focuses on the two DL SCs that are derived from two popular DL frameworks: TensorFlow and PyTorch. Here we give the formal definition of a DL SC. We define a DL SC as a directed graph $G =< V_{from}, V_{to}, E >$ starting from a root project (i.e., TensorFlow or PyTorch), where V_{from} is a set of downstream projects, V_{to} is a set of upstream projects imported by code files of downstream projects, and $E \subseteq V_{from} \times V_{to}$ is a set of directed edges representing import relationships. Compared with the previous studies defining SC as a package dependency network [28], we extend this definition by considering all dependent projects (not just packages) to depict an overall picture of SC. By collecting the data from World of Code (WoC) [29] and Libraries.io dataset[3], we build the SCs of TensorFlow and PyTorch. Based on these two DL SCs, we attempt to investigate the DL SCs from three aspects: structure, application domains, and evolutionary factors. We present three research questions and the main findings are as following.

RQ1: (Structure) *What is the structure of the DL SCs, how does it evolve, and how to identify the vulnerabilities in the structure?*
We find that the DL SCs exhibit a short and sparse hierarchy structure, with only five or six layers, and more than 80% of the projects are distributed in the second layer. The number of projects in the SCs shows an overall increasing trend. Projects often receive downstream projects shortly after the release of their packages, and then the growth rate becomes faster and tends to stabilize. We propose three criteria to evaluate the vulnerabilities of the DL SCs and indicate their potential risky projects.

RQ2: (Domain Distribution) *What domains do the DL SCs cover, and are the two SCs different?*
We find 51 types of packages involved in the two categories: domain related packages and non-domain related packages, and 26 types of projects covering four categories: application, research, learning, and software support. A comparison reveals the similarities and differences between the two SCs, e.g., TensorFlow SC provides a wealth of packages in experiment result analysis, while PyTorch SC contains more specific framework packages. Moreover, the distribution of project types fluctuates over time, e.g., the proportion of Raspberry Pi based APP is gradually increasing.

RQ3: (Evolutionary Factors) *What factors are related to the number of downstream projects in the DL SCs?*
We find that the most critical factor is the number of authors, which shows a nonlinear relationship with the number of downstream projects. The number of dependent DL packages displays a liner negative correlation. However, the number of stars, project age, and many SCs related factors, e.g., SC name and the level of package do not show significant relationship.

Overall, our study makes the following contributions:

- A first study on the hierarchical structure of the DL SCs revealing their structure, application domains, and evolutionary factors.
- Practical insights on the sustainable DL SCs for DL communities maintainers, DL practitioners, and researchers.
- An approach for constructing SCs and a dataset of the TensorFlow and PyTorch SCs containing 491,299 projects.

In the remainder of the paper, Section 2 introduces the dataset and the approach to construct DL SCs. Section 3 to Section 5 present methods and answers to each research questions. Section 6 discusses the implications. Section 7 presents threats to validity. Section 8 presents related work and Section 9 concludes the paper. A replication package of our analysis is available on https://github.com/SunflowerPKU/ICSE22_SC_Data.

2 DATASET AND SC CONSTRUCTION

2.1 Data Collection

We mainly use two sources for composing the dataset. For obtaining the package reference relationships, we use the latest version of WoC: version Q: 2019.12. WoC collects almost all the Git projects on the Internet and provides the API that can efficiently retrieval technical dependencies [29]. The technical dependencies have been extracted by parsing the content of all blobs related to several different languages and already stored in the specific files. Therefore, it is easy to obtain a list of commits and projects that imported a certain package by searching these files. The format of each piece of data returned as follows: *commit;repo;timestamp;author;blob;language; filename;model1...* We only focus on two domains: *repo* and *timestamp*. We find that there is little data in the last two months (i.e., 2019.11 and 2019.12) because it takes time for WoC to collect and build the data [29]. Therefore, we delete the last two months' data during analysis.

Libraries.io dataset tracks over 2.7m unique open source packages, 33m projects and 235m inter dependencies between them. We downloaded its latest version: January, 2020. We use this dataset to identify which projects in a layer of the SC have released packages and the information about these packages to build the SC.

To investigate RQ2 and RQ3, we extract README files and other attributes of the projects. Because some attributes are not included in WoC and Libraries.io, we obtain them through GitHub API.

[3]https://libraries.io/data

2.2 SC Construction

The structure of a DL SC is a network with nodes representing projects and edges representing package reference relationships. Algorithm 1 shows how we construct a DL SC. We start from a DL framework, i.e., TensorFlow or PyTorch, and search Libraries.io to obtain its related package. For many packages, their import names are different from packages' names (e.g., PyTorch's import name is torch) and such mapping information is unavailable. Therefore, we manually label the import names of the packages by reading their tutorials. Then, for each import name, we search WoC to obtain the downstream projects that import this package to build the next layer of the DL SC. We repeat this process until the projects in the last/highest layer do not have new packages released, or the packages have no downstream projects. The following points need to be noted. 1) One project may import a certain package multiple times. For this case, we only retain the first import time to represent the establishment time of the reference relationship; 2) For a package with multiple versions, its earlier versions may not import any package in the DL SC, but at time T, it imported. When we obtain the downstream projects of this package, we only retain the downstream projects whose import time is later than T. 3) When we need to analyze the hierarchical structure of SCs, a project may import the packages at different levels, making it difficult to determine the level of this project. For this case, we classify it as a higher-layer project. For example, if a project imports packages located at both the second layer and the third layer, we consider this project to be at the forth (3+1) layer.

Algorithm 1: Constructing DL SC

```
Input: $R_i$ ;                    // url of projects on the i-th layer
Output: DL SC
1  initialization: $i \leftarrow 1$ ;          // start from the 1st layer
2  $IN_i \leftarrow \{\}$ ;    // import names of the packages released by the
     projects on the i-th layer
3  $R_i \leftarrow \{url\ of\ TensorFlow\ or\ PyTorch\}$;
4  while true do
5      $IN_i \leftarrow \{\}$ ;
6      for $r \in R_i$ do
7          search Libraries.io to obtain the released package $p$;
8          if $p$ then
9              manually label the import name of $p$: $i\_n_p$;
10             $IN_i$.insert($i\_n_p$);
11     if $IN_i$ is null then
12         return ;                    // no packages are released
13     else
14         $i \leftarrow i + 1$, $R_i \leftarrow \{\}$ ;          // next layer
15         for $i\_n_p \in IN_i$ do
16             search WoC to obtain projects that import $i\_n_p$: $r$;
17             $R_i$.insert($r$);
18     if $R_i$ is null then
19         return ;                 // no downstream projects exist
```

3 RQ1: STRUCTURE

3.1 Methodology

Based on Algorithm 1, we build the SCs of TensorFlow and PyTorch. We depict the structure of two SCs and their evolution characteristics. Considering malicious actions may introduce malicious code directly in the end software or indirectly through dependent constraints [33], we propose three criteria to identify high-risk nodes by analyzing the SCs structure.

3.2 Results

3.2.1 Structure Characteristics. Table 1 lists the basic information of two DL SCs. Since the initial release time of TensorFlow is earlier, the length of the time range for constructing TensorFlow SC is 16 months longer than that of PyTorch. Both TensorFlow and PyTorch SCs have substantial projects, i.e., 355,392 and 136,507 respectively. However, these two SCs are short, with only six or five layers, which indicates that the length of SCs can not grow indefinitely. In terms of packages, only a few projects have released packages ($\frac{\#packages}{\#projects}$: TensorFlow: 0.29%; PyTorch: 0.51%), which means that almost all the projects in the DL SCs are use-oriented. Moreover, one problem seems to be that a significant number of these packages are not referenced by any project ($1 - \frac{\#projects\ with\ downstream}{\#packages}$: TensorFlow: 33.92%; PyTorch: 40.03%). From #import relationship, we can see that the structures of DL SCs are sparse. One possible reason is that few nodes have the ability to attract downstream projects, which is already confirmed above: there are few software packages in the SCs. Another possible reason is that the existing internal dependencies is simple.

To explore the second conjecture above, we analyze the dependencies of projects at different layers in the SC. As shown in Table 1, most of the projects (TensorFlow SC: 89.23%; PyTorch SC: 82.43%) are distributed in the second layer, i.e., directly dependent on TensorFlow or PyTorch. It means that TensorFlow and PyTorch are not only the starting points of the two SCs but also the key packages that contribute most of the reference relationships. Except for the first layer, the number of projects in each layer decreases with the increase of the number of layers, which indicates that with the increase of DL SC layer, it seems more difficult to attract downstream projects. We can see that for most of layers, instead importing different layers of packages to work together, projects seem to be more likely to only import packages published by the project immediately above them except the third layer of PyTorch SC. This is generally because the package has integrated the functions of the package it directly depends on. For example, project "*Face-Recognition-iOS-app-on-NBA-players*" (third layer) is an IOS APP that aims to recognize NBA players' faces.[4] Because it is deployed on the IOS platform, it imports "*tfcoreml*" (second layer), a package directly depending on TensorFlow to integrate machine learning models into IOS APP.[5] As for project "*LD-Net*"(third layer),[6] it imports both "*torch*"(first layer) and "*tensorboardX*" (second layer).

[4]https://github.com/liuyinhsiang/Face-Recognition-iOS-app-on-NBA-players
[5]https://developer.apple.com/documentation/coreml
[6]https://github.com/gonewithgt/LD-Net

Table 1: Basic Information of the DL SCs

	TensorFlow	PyTorch
Start Time	2015.11	2017.03
End Time	2019.10	2019.10
#Projects (nodes)	355,392	136,507
#Packages	1,022	697
#Import relationship (edges)	369,201	152,059
#Layers	6	5
#Projects with downstream	614	418

Table 2: Project Dependencies in the DL SC

TensorFLow			PyTorch		
Project Layer	**Layers of Dependent Projects**	**Count**	**Project Layer**	**Layers of Dependent Projects**	**Count**
1	/	1	1	/	1
2	1	317,115	2	1	112,516
3	2	23,323	3	2	8,072
	1,2	12,252		1,2	15,255
4	3	1,268	4	3	422
	1,3	500		1,3	28
	2,3	348		2,3	104
	1,2,3	321		1,2,3	81
5	4	211	5	4	24
	1,4	3		1,4	1
	2,4	3		3,4	3
	3,4	34			
	1,2,4	6			
	1,3,4	1			
	2,3,4	1			
	1,2,3,4	4			
6	5	1			

The latter only provides visualization during model training and contributes 48% of the projects in the third layer of PyTorch SC.[7]

3.2.2 Evolution Characteristics. Figure 1 shows the growth trend of TensorFlow SC and PyTorch SC. We can see that since the release of TensorFlow and PyTorch, the numbers of new projects added each month in their SCs show a growth trend overall. By May 2019, the number of new projects in TensorFlow and PyTorch SCs both reached a peak (TensorFlow SC: 16,518 projects per month; PyTorch SC: 8,836 projects per month). The growth of the software packages shows a similar trend.

We also analyze the growth trend of different layers of TensorFlow SC and PyTorch SC. For both SCs, the second and third layer projects appeared shortly (< 4 months) after the DL frameworks were released, while the fourth and subsequent layer projects did not appear until a year later. It indicates that for the packages in the first two layers of the SCs, once they are released, they may quickly attract their users. For example, just less than a month after the release of PyTorch, a package that directly relies on torch named "*block*" was released and aims to simplify the matrix operations

[7]https://pypi.org/project/tensorboardX

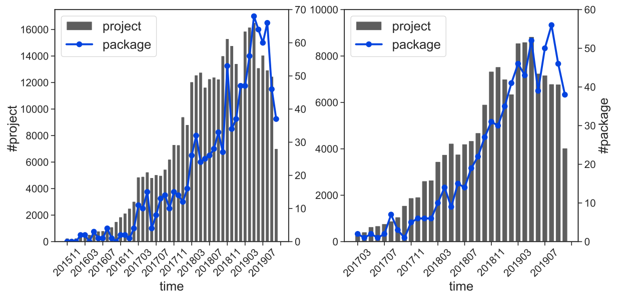

Figure 1: Number of New Projects/Packages Added in TensorFlow SC (left) and PyTorch SC (right).

involved in PyTorch.[8] After that, "*block*" has totally attracted 1,809 downstream projects and gradually formed its own SC.

Figure 2 shows the cumulative distributions of the fraction of time a project in a SC takes to have at least 10%, at least 50%, and at least 90% of its downstream projects. To facilitate analysis, we only consider the projects whose downstream projects is greater than or equal to five and age is greater than or equal to three months (TensorFlow SC: 549 projects; PyTorch SC: 370 projects). Specifically, the y-axis shows the fraction of projects that achieved 10%, 50%, and 90% of their downstream projects in a period of time that does not exceed the fraction of time shown in the x-axis. The cumulative changes of downstream projects in TensorFlow SC and PyTorch SC are similar. More than 70% of the projects have 10% of their downstream projects early, in 20% of their ages after the initial release (label A). We hypothesize that many of these initial downstream projects come from early adopters, who start using novel DL packages quickly after they are out. After this initial rising of popularity, the growth of half of the projects' downstream projects tends to stabilize, e.g., half of the repositories take 40% of their age to have 50% of their downstream projects (label B); and half of the projects take about 70% of their age to have 90% of their downstream projects (label C).

Figure 3 shows the distribution of the fraction of downstream projects obtained in the first and last month of the packages in the two SCs. In the first month, the fraction of downstream projects gained is 5.50%/13.32% (median/mean for TensorFlow SC) and 5.97% / 12.05% (median/mean for PyTorch SC). For the last month, the fraction of downstream projects gained is 11.99% / 16.22% (median / mean for TensorFlow SC) and 12.50%/16.27% (median / mean for PyTorch SC). By applying the Mann-Whitney U test [31], we find that for both SCs, the distributions are different (p-value < 0.001) with a medium effect size (TensorFlow SC: 0.27; PyTorch SC: 0.27). It means that for the packages in DL SCs, the downstream projects do not have explosive growth in the early stage. Compared to the early stage, the project accumulation in the later stage is faster, which is different from the accumulation law of the stars of GitHub projects [5]. For example, "*tvm*" is a DL compiler stack for cpu, gpu and specialized accelerators, which is in the second layer of PyTorch SC and was released on 2017-07-30.[9] During the first three months after its release, it attracted four downstream projects, and since

[8]https://github.com/bamos/block
[9]https://tvm.apache.org/

then this number increases to 12 projects per month. It indicates that the popularity of packages shows a steady and rapid growth.

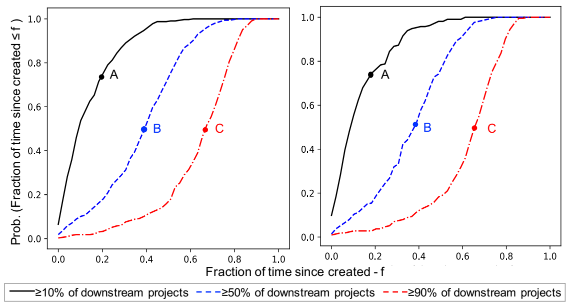

Figure 2: Cumulative Distribution of Downstream Projects in TensorFlow SC (left) and PyTorch SC (right).

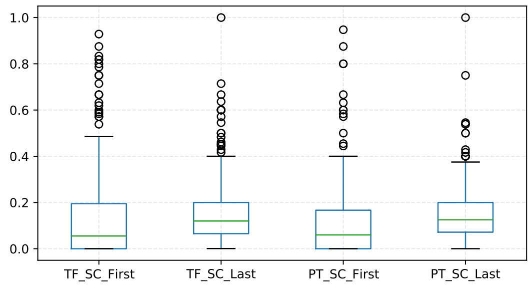

Figure 3: Fraction of Downstream Projects Obtained in the First and Last Month (TF: TensorFlow, PT: PyTorch).

3.2.3 Vulnerabilities. Through analyzing the structure of the DL SCs, we propose three criteria to help indicate possible risks.

1) *Most Impactive Projects.* The most critical projects have cascading impact on their downstream projects. Only analyze the number of incoming connections would not be enough to find the most dependent upon projects. Therefore, we also consider the transitive dependency when calculating in-degree. As shown in Figure 4, a substantial number of projects in the SCs do not have any downstream projects, which means that they have no chance to impact other projects. We also labelled the top three most impactive projects in the two DL SCs. We can see that the most impactive projects are TensorFlow and PyTorch. Besides, some supporting tools, e.g., "*tensorboardX*"[10] and "*picamera*"[11], are extremely popular so that they may introduce great threats to the security of the SCs. Therefore, careful maintenance by developers is needed.

2) *Most Vulnerable Projects.* The more packages a project depends on, the more vulnerable it may be [45]. Therefore, we calculate the out-degree distribution of the projects considering their transitive dependency. As shown in Figure 5, more than 80% of the projects only import one DL package, which means that they are relatively safe. However, there are still a few projects that import dozens of DL packages. Once any dependent package is attacked, it will be affected. For example, "*neuralLOGIC*"[12] aims to build a knowledge base of DL and totally imports 50 DL packages. An intriguing note is that the top three most vulnerable packages in the two DL SCs are the same because some packages are suitable for different DL ecosystems. It means that distinct SCs are not fully independent; rather, they're developing and evolving together. For example, "*onnxruntime-copy*"[13] depends on package "*onnx-coreml*"[14] that is used to convert ONNX models into Apple core ML format with no specific framework restrictions.

[10]https://github.com/lanpa/tensorboardX
[11]https://github.com/waveform80/picamera

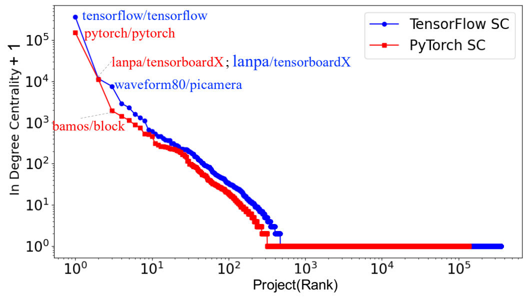

Figure 4: In-degree (Dependent Upon) Distribution of the Projects.

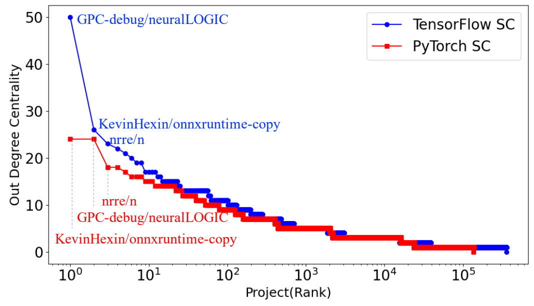

Figure 5: Out-degree Distribution of the Projects.

3) *Projects have the Most Chance of being a Single Point of Failure.* Projects may form clusters through inter dependencies. A few projects may become bridges among these clusters. Therefore, by identifying the nodes that have high betweenness centrality [42] but low degree centrality, we can find the projects that are more likely to be a single point of failure once they are removed [22]. As shown in Figure 6, "*gunpower*"[15] seems to have a more critical location in the PyTorch SC.

4 RQ2: DOMAIN DISTRIBUTION

4.1 Methodology

We answer this question from two perspectives: packages and projects. Software packages play a key role in the formation and evolution of SCs, so analyzing the functions of packages helps to understand the tool support provided by SCs. Considering many

[12]https://github.com/GPC-debug/neuralLOGIC
[13]https://github.com/KevinHexin/onnxruntime-copy
[14]https://pypi.org/project/onnx-coreml/
[15]https://pypi.org/project/gunpowder/

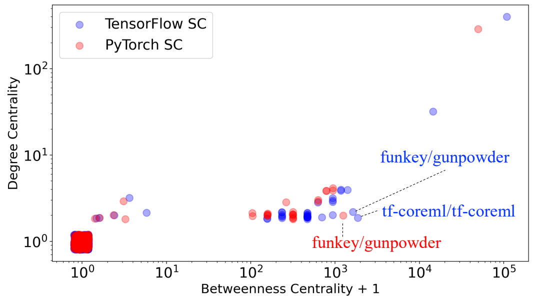

Figure 6: Degree Centrality and Betweenness Centrality of the Projects.

(40%) packages are trivial (no users) and analyzing all the packages is time-consuming, we decided to analyze the packages with no less than ten downstream projects (TensorFlow SC: 215 packages, PyTorch SC: 142 packages). We applied thematic analysis on their descriptions. It involves the following steps [9]: (1) initial reading of the descriptions, (2) generating the initial codes for each package description, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) designing the final themes. Steps (1) to (4) are performed independently by the first two authors. After this, a sequence of meetings is held to resolve conflicts and to assign the final themes (step 5). The final inter-rater reliability is 93.56%.

The application fields of projects in a SC can help us understand the status of SC evolution. To determine project types, we examine repository READMEs in the two SCs using Latent Dirichlet Allocation (LDA), a topic modelling technique widely used in Software Engineering (SE) research [1, 2, 20, 39]. We collect repository READMEs from WoC as following: a) retrieve the latest commit for each repository, b) obtain each repository's root folder structure from the tree object from the latest commit, c) check if "README.md" exists in the root folder structure, d) if contained, retrieve its content by its SHA-1 hash. Finally, we get 207,676 and 88,884 repository READMEs in TensorFlow SC and PyTorch SC respectively. We then perform the same data preprocessing steps on the two README datasets respectively to reduce the noise. We firstly remove code blocks marked with ‘. We then extract the first two sections of README as developers typically describe repository's functionality in the first or second sections based on our manual inspection of 383 randomly sampled READMEs (confidence level: 95%, margin of error: 5%). We also remove URL, numbers, punctuation and other non-alphabetical characters in the first two sections. We remove stop words using Long Stopword List provided by Ranks NL[16]. Finally, we use Ported Stemmer [36] to stem words. After removing preprocessed READMEs whose length less than 100 [40], 91,788 and 42,564 preprocessed READMEs remain in TensorFlow SC and PyTorch SC respectively. We then feed these two sets of preprocessed READMEs to LDA to identify project types.

LDA models a topic as a word probability distribution and a document as a topic probability distribution. Then it estimates topic-word distribution and document-topic distribution based on the word frequencies and word co-occurrences. The main challenge of applying LDA is to determine some optimal parameters including a) topic number k; b) iteration number I in estimation step; c) document-topic density α; d) topic-word density β as pointed in prior work [34]. We then use Genetic Algorithms (GA), which is widely used to optimize LDA parameters in SE researches [34, 39, 39] to determine the optimal parameters of LDA. GA searches optimal parameters by simulating the natural evolutionary process. We use Gensim's online LDA implementation and Pyevolve's GA implementation. For LDA, we set the parameter search space as: $k \in [2, 50], I \in [20, 2000], \alpha \in (0, 0.1], \beta \in (0, 0.1]$. We set all other parameters as default. For GA, we set both population size and generation to 100 to ensure sufficient configurations are tested and all other parameters as default following prior work [34, 39]. The optimal parameters for LDA on repository READMEs in TensorFlow SC is $k = 40, I = 1832, \alpha = 0.021, \beta = 0.081$ and the optimal parameters for LDA on repository READMEs in PyTorch SC is $k = 26, I = 1546, \alpha = 0.002, \beta = 0.007$. For each topic, we randomly selected ten READMEs for manual analysis to help us determine the meaning of this topic. To make topics easier to understand, we combine the same topics and conduct high-level abstractions.

4.2 Results

4.2.1 Packages in the DL SCs. Figure 7 and 8 show the packages distribution in the TensorFlow SC and PyTorch SC respectively, which are similar overall. We totally find 51 types of packages in the two DL SCs, covering domain-related packages (DR) and non-domain related packages (NDR). The non-domain related packages account for more than half, including framework,[17] model,[18] wrapper,[19] and tutorial, and provide a rich variety of support tools, including related tools to assist model training, analysis, and deployment. Domain-related packages cover a wide range of fields, such as hot areas including CV (computer vision), NLP (natural language processing), and RL (reinforcement learning), as well as interdisciplinary fields, e.g., biology, medicine, geography, and physics, under which there are two main package types: model and framework. It indicates that DL has been extended beyond the range of information field, not only limited to usage but also to provide service and support (i.e., packages).

There are also some differences between the two SCs. Firstly, from the perspective of packages type, PyTorch SC seems to contain more framework packages in the specific domains, while TensorFlow SC seems to contain more general supporting tools (e.g., Model Pruning, Encryption, and Interpretability). This differences may originate from that PyTorch delivers a more flexible environment and developers need to build every training loop, which is a better pick for a team that has a deeper understanding of DL concepts behind commonly used algorithms and develop their own framework [6]. As for TensorFlow, it has a large and well-established user base and a plethora of tools to help productionize DL, so there are plenty of outstanding tools. Secondly, from the perspective of

[16]https://www.ranks.nl/stopwords

[17]A framework is an ecosystem of tools and other resources that provide workflows with high-level APIs, which can help developers to build and deploy models, e.g., https://pypi.org/project/delira/.

[18]A model is a file that has been trained to recognize certain types of patterns, usually applicable to certain type of tasks, e.g., https://pypi.org/project/keras-transformer/.

[19]A wrapper aims to provide a different way to use wrapped object, e.g., providing a simpler interface, or adding some functionality, e.g., https://pypi.org/project/canton/.

packages proportion, TensorFlow SC provides a wealth of packages in experiment result analysis, represented by *TensorBoard*, whereas the support of PyTorch SC in this aspect is relatively weak. Further, PyTorch SC seems to have a higher proportion in general Framework, while TensorFlow SC seems to pay more attention to general wrappers. The reason is similar as mentioned before – PyTorch seems more flexible and developer-friendly, and TensorFlow feels a lot more like a library rather than a framework, so the wrappers aim to make it more simple to use and applicable to different scenarios. For the domain-related packages, both two DL SCs cover various fields. Among them, TensorFlow SC provides more packages in reinforcement learning (RL), which is mainly due to the popularity of *TensorFlow Agnent*[20] and *OpenAI Baselines*"[21].

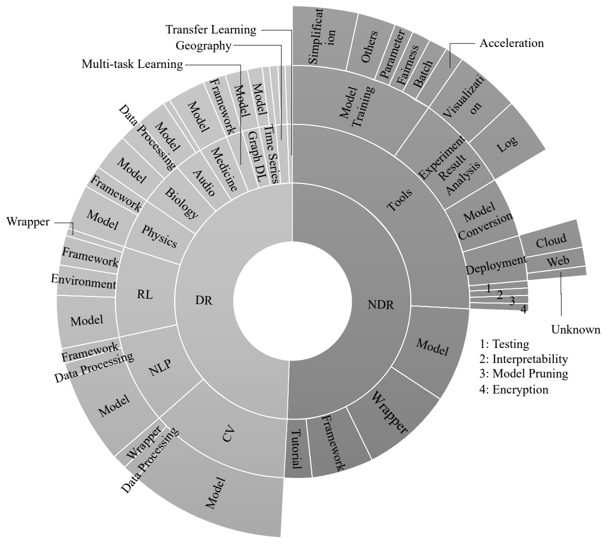

Figure 7: Packages Distribution of TensorFlow SC.

For each type of packages, we count the number of the projects that import them. Table 3 lists the top 10 most popular package types in the two SCs. For Both SCs, the non-domain related packages of the DL tools are most popular. Compared with PyTorch SC, the domain related packages of RL and non-domain related packages of wrappers in the TensorFlow SC are obviously more popular, consistent with the abundance of such packages. In the PyTorch SC, the non-domain related packages of DL framework has a wide range of downstream projects. For the packages in the hot fields, such as CV and NLP, their popularity does not show significant differences in the two DL supply chains.

4.2.2 Projects in the DL SCs. Figures 9 and 10 show the project types of the TensorFlow SC and the PyTorch SC respectively. We can see that for both SCs, their projects all cover four fields: research, application, learning, and software support. An obvious difference is that the TensorFlow SC has far more project types than that of the PyTorch SC: 23 VS 13. This may be due to the relatively small number of projects in the PyTorch SC.[22] For two SCs, their application categories contain rich types, both including the primary and important areas of DL: CV, NLP, and RL. TensorFlow SC also involves the areas such as *Self-driving* and *Robot Control*, which are closely related to the industry. This is because deployability in production environments is still TensorFlow's strength with *TensorFlow Serving* (Type: *Deployment*) being the most popular model server.[23] Moreover, *Raspberry Pi APP* is a noteworthy type, which has never been mentioned in previous studies [18]. It refers to the applications based on Raspberry Pi that is a series of small single-board computers (SBCs) to help people of all ages and knowledge levels explore the world of programming and computing.[24] This kind of application appears in the SC mainly because of a package named *TensorFlow Lite* that enables developers to run

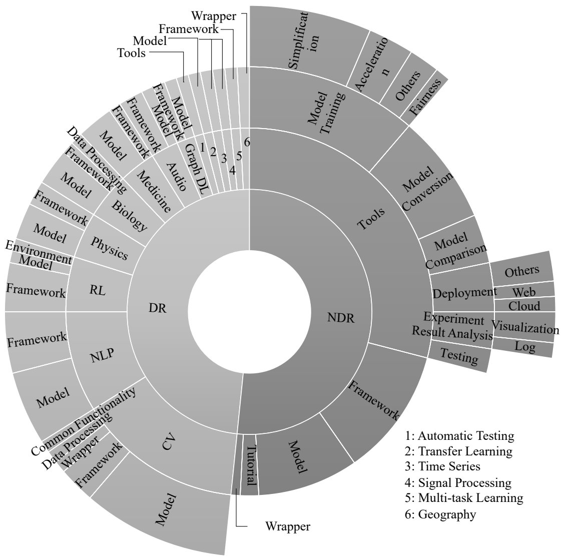

Figure 8: Packages Distribution of PyTorch SC.

Table 3: Top 10 Most Popular Package Types

	TensorFlow SC		PyTorch SC	
Rank	Type	Count	Type	Count
1	NDR, Tools	18,372	NDR, Tools	14,080
2	DR, RL	3,902	NDR, Framework	2,930
3	DR, CV	3,138	DR, CV	2,412
4	DR, NLP	1,437	DR, NLP	821
5	NDR, Wrapper	1,159	DR, Audio	782
6	NDR, Model	829	DR, RL	376
7	NDR, Framework	693	DR, Medicine	336
8	DR, Biology	619	DR, Physics	315
9	DR, Physics	563	NDR, Model	261
10	DR, Medicine	353	DR, Graph DL	242

[20]https://www.tensorflow.org/agents
[21]https://github.com/openai/baselines

[22]Note it cannot be assumed that the unidentified types do not exist in the SC, which may be because their number is small, the LDA algorithm cannot identify.
[23]https://github.com/tensorflow/serving
[24]https://www.raspberrypi.org/

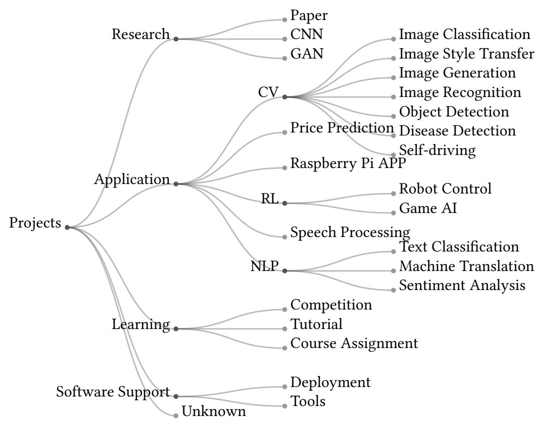

Figure 9: Project Types in the TensorFlow SC

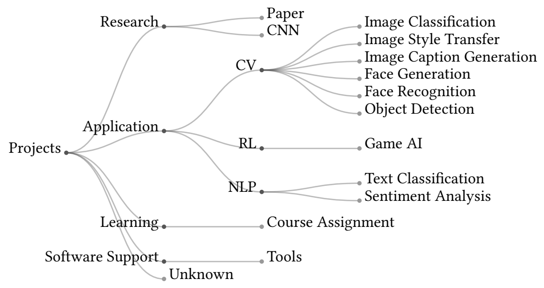

Figure 10: Project Types in the PyTorch SC

DL models quickly on mobile and embedded devices (such as Raspberry Pi) with low latency [25]. TensorFlow has a much bigger and established community behind it, so it is easier to find resources/ tutorials (Type: *Course Assignment, Tutorial*) to learn TensorFlow. It is also preferred by more DL competition participants although it is more difficult to start with, for which TensorFlow's high-level API Keras contributes a lot.[26]

We demonstrate the evolution of the project types distribution in the two SCs, as shown in Figures 11 and 12. It can be seen that the distribution of project types fluctuates over time, among which CV contributes a large proportion – about 40% (30%) of all the projects in the TensorFlow (PyTorch) SC. It indicates that CV has always been a hot field in the application of DL technology. One obvious difference is that the proportion of *Research* in the PyTorch SC is much higher than that of the TensorFlow SC: median: 32.3% vs 8.8%. This is because PyTorch is more tightly integrated with the Python language and is much easier to understand and debug [25]. Therefore, if developers want to create products related to artificial intelligence, TensorFlow is a good choice, whereas PyTorch is more suitable for research and building rapid prototypes. Similar to the packages, there are also more projects about *Learning Resources* and *Software Support* in the TensorFlow SC than the PyTorch SC. The proportion of *Raspberry Pi based APP* is gradually increasing, which means that smart robots and embedded DL projects are gradually attracting the attention of more and more researchers.

We also analyze the distribution of project types at different layers of the SCs (the results are shown in the appendix). Since the number of projects on the second layer of the SCs accounts for more than 80% of all the projects, the evolution of projects types on the second layer is similar to Figures 11 and 12. For subsequent layers, there are higher proportions of projects difficult to identify their types through LDA, which may be because a few new types of projects have appeared as the SCs extend. For example, *Maryam*, a project on the fifth layer of the TensorFlow SC, aims to provide a robust environment to harvest data from open sources and search engines quickly and thoroughly.[27]

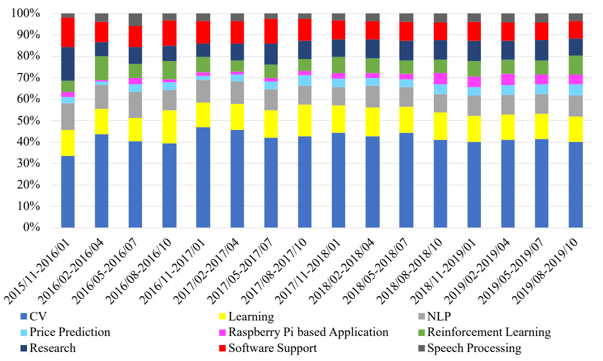

Figure 11: Evolution of the Project Types Distribution in TensorFlow SC.

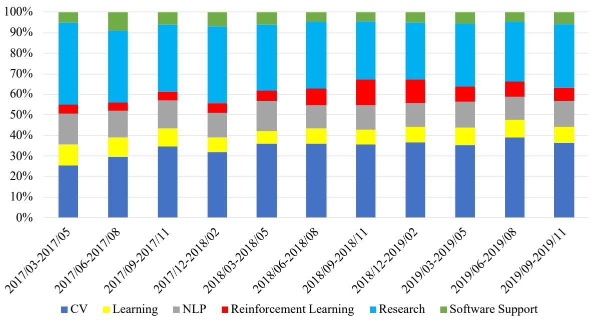

Figure 12: Evolution of the Project Types Distribution in PyTorch SC.

5 RQ3: EVOLUTIONARY FACTORS

5.1 Methodology

RQ1 suggests that the numbers of the downstream projects of packages distribute unevenly. Around 30% to 40% of the packages do not have downstream projects, which not only wastes the energy of package developers, but also hinders the sustainable development of SCs. Therefore, we try to understand the factors related to the popularity of the packages.

[25]https://www.tensorflow.org/lite/
[26]https://keras.io
[27]https://github.com/saeeddhqan/Maryam

We hypothesize that package domain is one of the important factors, so we analyze the manually labelled packages used to answer RQ2 to answer this RQ, i.e., the packages with no less than ten downstream projects (357 packages). We consider three categories of factors that may related to the number of downstream projects.

- **Related Project Feature**: *#commits*; *#contributors*; *age*; *#stars*
- **Package Domain**: *DR / NDR*; *sub_domain*
- **SC Feature**: *SC name (TensorFlow SC / PyTorch SC)*, *No.layer*, *#dependencies*

Because the relationships between response and predictors are not assumed to be linear, we fit Generalized additive models (GAM), which is applicable to analyze nonlinear and non-monotonic relationships between the response variable and explanatory variables [21]. We use the GAM function provided by the package "mgcv" in R language [37]. Firstly, because most of the variables are highly skewed, both response and numeric predictors are log-transformed. Secondly, we use the variance inflation factor (VIF) to analyze the multicollinearity of the prediction variables. Since the VIF of all variables is less than 5, there is no significant multicollinearity among all the variables [30]. Finally, we build GAM as Equation 1, where Y represents the number of downstream projects, f_i represents the smooth functions estimated by the model by maximum likelihood, $X_{1...5}$ represent the numeric predictors (e.g., *#commits*), $X_{6...9}$ are categorical variables (e.g., *SC name* and *No.layer*), and ϵ represents intercept. We first include all variables into the analysis, and then used the backward elimination method to gradually eliminate the insignificant variables through the p value (p < 0.05) [38].

$$g(Y) \sim f_1(X_1) + ... + f_5(X_5) + X_6 + ... + X_9 + \epsilon \tag{1}$$

5.2 Results

The final model has three predictors: *domain*, *#authors*, and *#dependencies*. The deviance explained is 41%, which means the factors that related to the number of downstream projects are complicate and our model can explain part of them. However, this model is acceptable because our aim is to understand the relationships between the independent variables and dependent variable not to predict the dependent variable. Tables 4 and 5 show the fitting results. The Figures of the fitting results of factor smoothing function are shown in the appendix. All the variables are significant except *domain: NDR*. The results indicate that these three predictors are related with *#downstream_projects* of the packages. The order of edf of two variable smoothing functions is: $s(\#authors) > s(\#dependencies)$, which means that the influence of *#authors* is the more critical than *#dependencies*. The influences of *#authors* on *#downstream_projects* is nonlinear. When *#authors* is small, it is negatively related to *#downstream_projects*, whereas when *#authors* reaches a certain degree (> 7 developers), it shows a significantly positive relationship. This may indicate that for small teams, the collaboration among team members may reduce the efficiency of developers. The relationship between *#dependencies* and *#downstream_projects* is linear, and *#dependencies* is negatively related with *#downstream_projects*. Dey et al. [15] obtain the similar finding: packages having fewer packages as dependencies are likely to see an increase in downloads. However, the number of stars, age, and many SCs related factors, e.g., *SC name* and *No.layer* do not show significant relationship.

Table 4: Parameter Estimates

Parameter	Estimate	Std Err.	t	Pr > |t|
Intercept	3.197	0.244	13.092	<2e-16
domain-related (DR)	0.558	0.262	2.126	0.035
non-domain related (NDR)	0.474	0.263	1.807	0.072

Table 5: Fit Summary for Smoothing Components

Smooth Terms	edf	Ref.df	F	p-value
s(#authors)	4.101	4.976	33.23	< 2e-16
s(#dependences)	1.809	2.287	5.80	0.003

6 IMPLICATIONS

Our study has rich implications for DL community maintainers, DL practitioners, and researchers.

DL Community Maintainers. Our findings reveal the evolutionary structure and risks of DL SCs, which can provide insights for maintainers to help the healthy and sustainable development of the DL communities. Through modeling the structure of DL SCs, we find that it is a short sparse hierarchical structure (RQ1). Making this structure explicitly visible can help DL community maintainers understand the scale and status of the DL SCs, which can bring insights for formulating targeted development strategies, e.g., when the growth rate suddenly drops, give an early warning. The criteria such as "most vulnerable projects" can warn the community maintainers and project owners which links and nodes are susceptible to disruptions in order to take precautions to ensure SC security. We also discovered that the top three most vulnerable projects are the same in both TensorFlow and PyTorch SCs, implying that collaboration between different DL communities is required and efficient in order to ensure the safety of the SCs.

For RQ2, we carefully compare the packages and projects in the TensorFlow SC and PyTorch SC and obtain a series of characteristics of these two SCs. For example, Raspberry Pi based Application is only observed in the TensorFlow SC. This difference is partly due to their superiority, e.g., TensorFlow SC is superior in on-device inference. The differences we reveal can help maintainers better understand what they need to optimize to remain competitive. We believe that the two DL frameworks will constantly impact each other, mimic their most effective solutions while improving existing deficiencies. In the near future, the two DL SCs may become much similar to each other.

For emerging DL frameworks, we explore the SCs of the two well-known DL frameworks depicting the structure, application domains, and evolutionary factors, which can serve as examples to lead their own ecosystems. For example, maintainers can model their SCs, analyze the evolution, and compare with TensorFlow and PyTorch SCs, so as to determine their directions.

DL Practitioners. DL practitioners can benefit from our study from the following two aspects. On the one hand, DL practitioners can choose the suitable DL framework more easily. TensorFlow and PyTorch both are open source DL frameworks extensively used in academic research and commercial code. Therefore, it may be difficult for beginners to choose. Our study carefully compare these two DL SCs, which can help practitioners determine which choice is best for their projects. Here, we demonstrate several differences based on the results of RQ2 to help DL practitioners make choices. **1) Ease of use.** We find that the PyTorch SC contains many frameworks and is preferred by researchers, whereas the TensorFlow SC provides various wrappers. It is mostly because PyTorch provides a relatively low-level environment that allows more freedom in writing customized layers, is more tightly integrated with Python, and is easier to debug, whereas TensorFlow is frequently criticized for being difficult to understand and use [32]. Fortunately, the community was aware of this and released TensorFlow 2.0 in Oct. 2019. **2) Deployment.** We discover that there are more embedded applications in the TensorFlow SC, for example, Raspberry Pi-based applications, which is due to TensorFlow's capacity to be deployed on various platform (packages: *TensorFlow Lite* and *TensorFlow Serving*), allowing its users to develop different types of systems and used in production environments. However, *PyTorch Mobile* is the only mobile support in the PyTorch SC. **3) Experiment results analysis.** TensorFlow SC provides a tool called *TensorBoard* that can enable practitioners to track metrics, visualize the model graph, view histograms of weights, biases, or other tensors as they change over time, and much more. Although there are the integrations of *TensorBoard* with PyTorch (i.e., *TensorBoardX*), it is not supported natively. **4) Community Support.** Figures 7 and 8 show that both TensorFlow and PyTorch SCs provide abundant supporting tools. However, from the perspective of production, the TensorFlow community is much larger and more active, whereas PyTorch is more suitable for research. As shown in Figures 11 and 12, TensorFlow SC has more resources in terms of tutorials and competitions because PyTorch is a relatively newer framework.

On the other hand, our findings can provide guidance for package developers in the DL SCs. Firstly, the results of RQ1 illustrate the cumulative characteristics and real-time changes of the number of the downstream projects of packages, which can be the reference for the maintainers to understand the application of their packages, so as to timely analyze the reasons and improve the competitiveness of the packages. Secondly, the results of RQ2 reveal the distribution of different types of packages in the two SCs, which can provide a reference for developers to develop their own packages, e.g., avoid developing duplicate packages. Thirdly, the results of RQ3 reveal some factors related to the number of downstream projects of packages, which sheds light on developing popular packages. For example, it is best not to rely on too many upstream packages.

Researchers. Researchers can build on our findings and ask new questions in order to have a deeper understanding of software SCs and perform relevant research. The method we propose for constructing DL SC is not confined to the field of DL. As a result, researchers can use this algorithm to construct SCs of other fields and study their structure, growth, and evolution. In RQ1, considering transitive dependencies, we put forward three criteria for identifying potential vulnerabilities. Although they look straightforward, they provide a basis for risk mitigation from SC perspective. Future work can build on our metrics and combine risk-related factors(e.g., project health) to refine these metrics and investigate mechanisms of risk management. Our research has provided a basic exploration of the DL SCs, and further research might be conducted in the future based on this foundation. For example, we observe that projects are more likely to import packages released by the project immediately above them for most of the layers in the DL SCs. Analyzing the origins of this phenomenon can help us grasp what software dependence is really about. In RQ2, we find that it is challenging when we use LDA to identify the types of projects on higher layers (>3), which might be due to the emergence of a few new types of projects as the SCs expand. As a result, a manual label may be required to assist us to comprehend the breadth of the DL SCs. We discover in RQ3 that the factors affecting the number of downstream projects are complicate. Future research can look into other critical aspects that can help us better understand how software SCs develop and evolve. Besides, future research can also investigate the reasons for the deprecation of frameworks from the perspective of their SCs.

7 THREATS TO VALIDITY

Internal validity concerns the threats to how we perform our study. The first threat relates to the method for constructing DL SC. To obtain the projects in the next layer, we need to known the import names of the packages in this layer. We obtain this information through reading the tutorials and source code of the packages. In order to ensure the correctness of the labels, the first two authors annotate independently, and then discuss the inconsistent cases (21/1,719) to obtain a consensus. The second threat relates to the thematic analysis applied for determining the functions of the packages (Section 4.1). To mitigate this threat, the initial selection of themes is performed individually by the first two authors. Then, we compare our list of codes and themes after this initial proposal and create a coding guide with definitions and examples for each identified theme. The coding guide is then used by each author to independently examine the whole collection of data. The coding guide is redefined based on feedback from the second round of analysis, and the data is independently evaluated for the third and final time. The last threat comes from the LDA algorithm used to identify the projects' types (Section 4.1). The choice of parameters is a potential threat. To minimize this threat, we use GA to tune LDA parameter with coherence score to measure LDA fit following prior SE researches [1, 19, 20, 34, 39, 39]. Another threats come from the process of labelling LDA topics. To minimize the threats, the first two authors perform open card sort separately and inconsistencies (6/66) are discussed until reaching an agreement.

External validity concerns the threats to generalize our findings. When conducting this study, we only focus on the SCs that originate from the two prominent DL frameworks. We choose them because DL is presently playing an important role in both academia and industry, allowing us to obtain timely and valuable results for practice and research. Most of our findings and methods can also be generalized to other software SCs. For example, the method for constructing DL SC and the criteria for identify vulnerabilities provide a nice starting point for analyzing other software SCs.

8 RELATED WORK

We discuss the related work of software SC from two aspects: 1) characteristics of dependency network; 2) risk and maintenance of software SC.

Characteristics of Dependency Networks. Recent years, software dependency network has gradually attracted researchers' attention. Prior work mostly investigate the packages in different registries, e.g., Npm and PyPI. A first large-scale analysis of the Npm ecosystems was conducted by Wittern et al [44]. Through analyzing the topology of popular Npm JavaScript libraries, they find that Npm packages are largely dependent on a core set of libraries. German et al. [17] also obtain the similar results by analyzing the packages in the R ecosystem. Besides, they find that user-submitted packages is the main reason for the growth of the ecosystem but the growth process is slow. Based on the analysis of Apache projects, researchers find that there is a exponential relationship between the number of projects and the dependencies among them [35].

Dey et al. [15] try to use the information of dependency structure to predict change of popularity of Npm packages. They find that the count and downloads of upstream and downstream runtime dependencies have a strong effect on the change in downloads. Our RQ3 also aims to investigate the related factors that influence the popularity of the packages. The difference is that instead of the number of downloads, we use the number of downstream projects as an indicator for popularity, which is closer to essence. Moreover, we consider other important factors, e.g., domain factors. Their follow-up work [14] studies the patterns of effort contribution and demand based on the Npm network. They find that users contribute and demand effort primarily from packages that they depend on directly with only a tiny fraction of contributions and demand going to transitive dependencies.

Some studies focus on more than one ecosystem and try to explore their differences. Decan et al. [10] compare the typologies of Npm, PyPI, and CRAN and find that different ecosystems have different properties, e.g., Npm is more interconnected than CRAN. Therefore, they claim that the findings obtained from one ecosystem can not always be generalized to other ecosystems. Kikas et al. [24] analyze the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems. They also find that there are significant differences across language ecosystems. However, our study indicates that, while there are some variances in application fields, the two DL SCs are similar in their structures.

In one most relevant study [18], the authors conduct a first study on the dependency networks of DL libraries. They study the project purposes, application domains, dependency degrees, update behaviors of DL projects. A serious problem is that it constructs dependent network via the "used by" mechanism provided by GitHub.[28] This mechanism makes a large number of downstream projects missed because it only supports the projects that have defined their dependencies in certain manifest files, e.g., *requirements.txt*. As a result, they only identify 46,930 projects in the TensorFlow SC, whereas we find 355,392 projects. They also do not consider the hierarchical structure of the SCs. Instead, our study analyzes the real package reference data and constructs the SCs layer by layer.

Risk and Maintenance of Software SC. There are substantial studies that focus on the risk of SCs and seek practices for maintenance. Through an interview, Bogart et al. [4] aim to understand how dependencies in R and Npm maintained. They find that developers often use ad-hoc mechanisms to negotiate change instead of existing awareness mechanisms due to information overload. Bavota et al. [3] investigate the timing of dependencies update and find that dependencies are updated only if major new features or bug fixes are released for the dependencies. By analyzing the usage of dependency version specification, Decan et al. [11] find that current tools and versioning schemes can introduce resiliency issues to the ecosystem. Cox et al. [8] measure dependency freshness in 75 different closed source projects and find that projects using outdated dependencies four times as likely to have security issues as opposed to projects that are up-to-date.

There are many studies focusing on the vulnerabilities or bugs in the SC. Decan et al. [12] analyze 400 security reports in the npm dependency network and investigate how and when these vulnerabilities are discovered and fixed, and to which extent they affect other packages in the packaging ecosystem in presence of dependency constraints. Ma et al. [27] investigate how developers fix cross-project correlated bugs and reveal the common practices of developers and the various factors in fixing cross-project bugs. Unlike the above studies, our study does not focus on the practices related to vulnerabilities/ bugs fix but focuses on identifying the high-risk nodes in the SCs to plan ahead.

9 CONCLUSION

Since the release of famous DL frameworks, such as TensorFlow and PyTorch, they have been widely applied in both academia and industry. Through dependency, a huge number of DL projects form a complicated DL SC. Understanding the structure and evolution of this SC can bring insights for guiding the healthy development of the DL ecosystem. In this paper, we focus on two DL SCs: TensorFlow SC and PyTorch SC. Firstly, we build these two SCs and analyze their structures, evolution, and vulnerabilities. We find that although the SCs contain hundreds of thousands of projects, there are only five or six layers. For most of the packages, they often attract their downstream projects at an early stage, after which their growth rates become faster and tend to stabilize. We defined three criteria that can indicate risky projects in the SCs. Secondly, we investigate and compare the types of packages and projects distributed in the two SCs. We find that there are 51 types of packages and 26 types of projects. The similarities and differences between these two SCs are identified. For example, although TensorFlow SC offers a wealth of learning resources and software support, and excels in embedded DL models, PyTorch SC is preferred among academics due to its ease of use. Finally, we fit the GAM model to examine the factors related to the popularity of the packages. We discover that the number of downstream projects is negatively related to the number of dependent packages, but the relationship with the number of authors is nonlinear. Our study is the first exploratory investigation concentrating on the hierarchical SC system that arose from a single project. Our findings preliminarily open the "black box" of DL SCs, putting forward practical insights and allowing for multiple paths of future research.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China Grants 62141209, 61825201.

[28]https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/exploring-the-dependencies-of-a-repository

REFERENCES

[1] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. 2020. Challenges in Chatbot Development: A Study of Stack Overflow Posts. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 174–185. https://doi.org/10.1145/3379597.3387472

[2] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going big: a large-scale study on what big data developers ask. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 432–442. https://doi.org/10.1145/3338906.3338939

[3] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *2013 IEEE International Conference on Software Maintenance*. 280–289. https://doi.org/10.1109/ICSM.2013.39

[4] Christopher Bogart, Christian Kästner, and James Herbsleb. 2015. When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 86–89. https://doi.org/10.1109/ASEW.2015.21

[5] Hudson Borges, André C. Hora, and M. T. Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), 334–344.

[6] Budek, Konrad and Tautkute-Rustecka, Ivona . 2020. PyTorch vs. TensorFlow – a Detailed Comparison. https://www.tooploox.com/blog/pytorch-vs-tensorflow-a-detailed-comparison [Online; accessed 27-July-2021].

[7] Caffe2 Community. 2021. Caffe2 and PyTorch join forces to create a Research + Production platform PyTorch 1.0. https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html [Online; accessed 23-July-2021].

[8] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 109–118. https://doi.org/10.1109/ICSE.2015.140

[9] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284. https://doi.org/10.1109/ESEM.2011.36

[10] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops* (Copenhagen, Denmark) *(ECSAW '16)*. Association for Computing Machinery, New York, NY, USA, Article 21, 4 pages. https://doi.org/10.1145/2993412.3003382

[11] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2–12.

[12] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

[13] Li Deng and Dong Yu. 2014. Deep learning: methods and applications. *Foundations and trends in signal processing* 7, 3–4 (2014), 197–387.

[14] Tapajit Dey, Yuxing Ma, and Audris Mockus. 2019. Patterns of Effort Contribution and Demand and User Classification Based on Participation Patterns in NPM Ecosystem *(PROMISE'19)*. Association for Computing Machinery, New York, NY, USA, 36–45. https://doi.org/10.1145/3345629.3345634

[15] Tapajit Dey and Audris Mockus. 2018. Are Software Dependency Supply Chain Metrics Useful in Predicting Change of Popularity of NPM Packages?. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering* (Oulu, Finland) *(PROMISE'18)*. Association for Computing Machinery, New York, NY, USA, 66–69. https://doi.org/10.1145/3273934.3273942

[16] Xuedan Du, Yinghao Cai, Shuo Wang, and Leijie Zhang. 2016. Overview of Deep Learning. In *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*. IEEE, 159–164.

[17] Daniel M. German, Bram Adams, and Ahmed E. Hassan. 2013. The Evolution of the R Software Ecosystem. In *2013 17th European Conference on Software Maintenance and Reengineering*. 243–252. https://doi.org/10.1109/CSMR.2013.33

[18] Junxiao Han, Shuiguang Deng, David Lo, Chen Zhi, Jianwei Yin, and Xin Xia. 2020. An Empirical Study of the Dependency Networks of Deep Learning Libraries. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 868–878. https://doi.org/10.1109/ICSME46990.2020.00116

[19] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do Programmers Discuss about Deep Learning Frameworks. *Empir. Softw. Eng.* 25, 4 (2020), 2694–2747. https://doi.org/10.1007/s10664-020-09819-6

[20] Mubin Ul Haque, Leonardo Horn Iwaya, and Muhammad Ali Babar. 2020. Challenges in Docker Development: A Large-scale Study Using Stack Overflow. In *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*, Maria Teresa Baldassarre, Filippo Lanubile, Marcos Kalinowski, and Federica Sarro (Eds.). ACM, 7:1–7:11. https://doi.org/10.1145/3382494.3410693

[21] Trevor J Hastie and Robert J Tibshirani. 2017. *Generalized additive models*. Routledge.

[22] Sarika Jalan and Camellia Sarkar. 2017. Complex Networks: an emerging branch of science. *Physics News* 47 (2017), 3–4.

[23] Jeff Hale. 2020. Is PyTorch Catching TensorFlow? https://towardsdatascience.com/is-pytorch-catching-tensorflow-ca88f9128304 [Online; accessed 27-July-2021].

[24] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, 102–112. https://doi.org/10.1109/MSR.2017.55

[25] Kurama, Vihar. 2021. PyTorch vs. TensorFlow: Which Framework Is Best for Your Deep Learning Project? https://builtin.com/data-science/pytorch-vs-tensorflow [Online; accessed 27-July-2021].

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *nature* 521, 7553 (2015), 436–444.

[27] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How Do Developers Fix Cross-Project Correlated Bugs? A Case Study on the GitHub Scientific Python Ecosystem. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 381–392. https://doi.org/10.1109/ICSE.2017.42

[28] Yuxing Ma. 2018. Constructing Supply Chains in Open Source Software. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 458–459.

[29] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 143–154. https://doi.org/10.1109/MSR.2019.00031

[30] Edward R Mansfield and Billy P Helms. 1982. Detecting multicollinearity. *The American Statistician* 36, 3a (1982), 158–160.

[31] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[32] Nicolas D. Jimenez. 2021. TensorFlow Sucks. https://nicodjimenez.github.io/2017/10/08/tensorflow.html [Online; accessed 23-July-2021].

[33] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. *CoRR* abs/2005.09535 (2020). arXiv:2005.09535 https://arxiv.org/abs/2005.09535

[34] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 522–531. https://doi.org/10.1109/ICSE.2013.6606598

[35] Sebastiano Panichella. 2014. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Software Engineering* (06 2014).

[36] Martin F Porter. 1980. An algorithm for suffix stripping. *Program* (1980).

[37] R documentation contributors. 2021. GAM: Generalized Additive Models with Integrated Smoothness Estimation. https://www.rdocumentation.org/packages/mgcv/versions/1.8-36/topics/gam [Online; accessed 27-July-2021].

[38] Willi Sauerbrei, Aris Perperoglou, Matthias Schmid, Michal Abrahamowicz, Heiko Becher, Harald Binder, Daniela Dunkler, Frank E Harrell, Patrick Royston, and Georg Heinze. 2020. State of the art in selection of variables and functional forms in multivariable analysis—outstanding issues. *Diagnostic and prognostic research* 4, 1 (2020), 1–18.

[39] Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. 2017. Cataloging GitHub Repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, EASE 2017, Karlskrona, Sweden, June 15-16, 2017*, Emilia Mendes, Steve Counsell, and Kai Petersen (Eds.). ACM, 314–319. https://doi.org/10.1145/3084226.3084287

[40] Christoph Treude and Markus Wagner. 2019. Predicting good configurations for GitHub and stack overflow topic models. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 84–95. https://doi.org/10.1109/MSR.2019.00022

[41] Athanasios Voulodimos, Nikolaos Doulamis, George Bebis, and Tania Stathaki. 2018. Recent developments in deep learning for engineering applications.

[42] Wikipedia contributors. 2021. Betweenness centrality — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Betweenness_centrality&oldid=1034163048 [Online; accessed 23-July-2021].

[43] Wikipedia contributors. 2021. Microsoft Cognitive Toolkit. https://en.wikipedia.org/wiki/Microsoft_Cognitive_Toolkit [Online; accessed 23-July-2021].

[44] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 351–361.

[45] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*. Springer, 95–110.

[46] Minghui Zhou, Yuxia Zhang, and Xin Tan. 2019. Software digital sociology. *SCIENTIA SINICA Informationis* 49, 11 (2019), 1399–1411.