Why do Researchers Prefer to use PyTorch Over TensorFlow?

Mohamed Ayman Mohamed
mamoham3@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

Steven Heung
syheung@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT

Deep learning has achieved significant success across diverse domains, which encouraged a lot of researchers to release various frameworks to enhance usability and performance for those engaged in deep learning research. Pytorch and Tensorflow are two very popular deep learning frameworks developed by Meta and Google, released in 2015 and 2016 respectively. Despite Pytorch being released at a later date, it has gained significant traction and has eventually become more widely used than Tensorflow by researchers. To help software developers characterize the important features that researchers do need in libraries in the future, we provide in-depth analysis for both frameworks.

Motivated by the research conducted by Jia et al., who scrutinized bug identification in TensorFlow, and Yin Ho et al., who delved into bug identification in PyTorch while comparing the outcomes with TensorFlow, our investigation centers on assessing the usability and evolutionary aspects of bugs in both frameworks. To ensure that our investigation is as comprehensive as possible, we analyze six client model codes from and all APIs of both Frameworks. Results show that PyTorch requires less effort for model implementation than TensorFlow. The PyTorch research community's rapid growth surpasses that of TensorFlow, facilitating quicker issue resolution. Additionally, PyTorch documentation is more detailed than TensorFlow.

KEYWORDS

Deep Learning Frameworks, Library features, Documentation Quality, Usability Functionality, Empirical Study

1 INTRODUCTION

The advent of deep learning has marked significant achievements across diverse domains, including computer vision, speech recognition, and natural language processing (NLP). The magic of deep neural networks (DNNs) within this paradigm, coupled with the abundance of big data, has been instrumental in driving this success. The continual progression of deep learning has been characterized by a discernible trend towards augmenting the depth of these networks. This evolutionary trajectory has led to a transformative shift away from conventional architectures, such as convolutional neural networks (CNNs)[57] and recurrent neural networks (RNNs)[66], towards more sophisticated structures like Transformers and graph neural networks (GNNs)[80]. This evolution, in turn, has proven the ability to address increasingly complex challenges, particularly within domains like Medicine[28] and self-driving[23].

Instead of reinventing wheels by implementing deep learning models from scratch, programmers often prefer to build their applications on mature libraries. various deep learning frameworks, such as Caffe by UC Berkeley[47], TensorFlow by Google[19], PyTorch by Facebook[58], and CNTK by Microsoft[65], offer mature and widely used options for building applications. Currently, PyTorch and TensorFlow are the two most popular frameworks. TensorFlow and PyTorch differ in their design philosophies, specifically in terms of static versus dynamic computation graphs. TensorFlow adheres to the paradigm of treating data as code and embracing the code as data idiom. This approach involves constructing a computation graph before its execution begins. In contrast, PyTorch adopts a more dynamic programming model, allowing users to define and execute graph nodes in tandem with the ongoing execution process.

Despite Pytorch being released at a later date, it has gained significant traction and eventually became more widely used than Tensorflow by researchers. According to Google search trends,pytorch has overtaken tensorflow in terms of search popularity at around 2021[3]. Specifically, In academia, the implementation of research papers in 2023 overwhelmingly favors PyTorch over TensorFlow, with a substantial ratio of 10:1[7]. Also, researchers predominantly favor pre-trained models, particularly evident on Hugging Face, a prominent platform for deep learning models. Notably, PyTorch-exclusive models constitute approximately 92%, up from 85 % in 2023, while TensorFlow models amount to only 8%, down from 16% 2022. In 2022, over 45 thousand PyTorch-exclusive models were added, compared to approximately 4 thousand TensorFlow models [21].

While previous studies, exemplified by Hulin DAI et al.'s[32] and Arpan et al.'s[45], endeavored to conduct analytical and experimental analyses aimed at unraveling the comparative potential usage of both PyTorch and TensorFlow in terms of performance on single GPUs with diverse models, these investigations fell short of providing a definitive explanation for the surging preference for PyTorch. The literature gap is notable in that some researchers have concentrated on performance aspects, while others have directed their attention towards understanding the root causes of bugs. Yet, none have systematically delved into comprehending the broader reasons for the preference for PyTorch. To **address this existing void in the literature**, the present paper is motivated to undertake an in-depth analysis from our empirical results, seeking to comprehend the underlying reasons contributing to the growing popularity of PyTorch over TensorFlow within research communities. This research trajectory is pivotal in offering insights that can guide programmers, who are tasked with implementing libraries, in prioritizing aspects aligning with researchers' preferences for one library over another.

To understand why researchers might choose PyTorch over TensorFlow, we need to explore the following research questions:

- **RQ1**: Which framework facilitates seamless utilization of APIs?
- **RQ2**: How does the prevalence of bugs in each framework influence their respective popularity over time?

Due to time restriction in the course, we would quantitatively measure APIs in both client code and documentation of both frameworks. Six models implemented using PyTorch and TensorFlow were scrutinized to quantify the density of APIs in each model. This quantitative analysis involved manual calculations. The examination relied upon the implementation details provided by Google[35], Nvidia[30], and HuggingFace[75]. In the context of documentation, six metrics, elucidated in the methodology sections, were employed to gauge the feasibility of APIs in both frameworks. Regarding the investigation into the second research question, the issues associated with the implementations of PyTorch and TensorFlow were mined from their respective GitHub repositories. The prevalence of bugs was discerned by utilizing logistic regression to classify the issues as buggy. Subsequently, a statistical analysis was conducted on the resultant graphs, as explicated in the results section.

Our contributions in this paper are as follows:

- Thoroughly investigate the APIs in both frameworks, analyzing client code and documentation using quantitative metrics for a comprehensive understanding.
- Thorough examination of the evolution over bugs for both frameworks and understanding their respective popularity over time.
- We offer a replication package for our code, providing a valuable resource for researchers seeking to reproduce and build upon our work.

2 RELATED WORK

2.1 Library / Framework comparisons

To the best of our knowledge, this is the first paper that directly compared PyTorch and TensorFlow under the perspective of a researcher under multiple criteria. The idea of comparing 2 libraries or frameworks is not a new concept. For instance, Ghimire [37] has compared two popular Python web frameworks: Flask [2] and Django [1], while Hafeez et al. [39] have compared two popular Python visualization libraries: Matplotlib [5] and Seaborn [9]. These papers have outlined the basis for criteria we need to compare, such as usability, understandability, documentation quality, and community support. However, we observed that very little work has been done that directly compared TensorFlow and PyTorch under the perspective of a researcher. Ho. et al [41] and Chen et al. [27] has directly compared the type of defects that occurred within the two libraries, but has not elaborated on how these defects affected researchers' decisions when choosing between the two libraries. While there are other sources [73] [48] that suggested PyTorch is superior in all of the aspects proposed by Ghimire [37] and Hafeez et al. [39], they did not back up their claims with supporting evidence nor measurements. This paper aims to verify these claims using our methodology explained below.

2.2 Comparing Bugs in PyTorch and TensorFlow

Jia et al. [46] have looked into the symptoms and cause of bugs inside the TensorFlow, while Ho. et al. [41] looked into PyTorch. They provide the method of classifying github issues into categories (bugs, feature requests, support, etc.), as well as related features that are relevant when classifying (Issue title, issue body, issue labels, etc.). They also provided high quality data sets which are useful for training our issue classifier.

Our work further expanded on those in multiple ways. First, we have tried a variety of different models and checked for validation accuracy to determine the best model for classifying issues. Second, we experimented with different input features (Issue title, Issue Body, etc.) and determined which features are the most effective when classifying whether an issue describes a bug or not. Third, we have created a complete list of bug related issues, taking all GitHub issues into account, instead of just a subsection of sampled issues. Lastly, we also tracked the issue creation date and issue closed date (if it exists), so that we can track how the number of bugs in PyTorch and Tensorflow evolved over time. This will be further elaborated in the methodology section.

2.3 Code Readability and Understandability

Scalabrino et al. [62] have proposed several objective measures that aim to measure the readability of code snippets. These metrics are based on previous researches into the properties of readable code, such as how full-name identifiers can ease code comprehension [51], and how the lack of cohesion negatively impacts the code quality [52] [72]. These ideas are further expanded by Dantas et al. [33] by introducing the idea of understandability score, which is based on SonarQube's cognitive complexity score [26].

While these metrics focus on evaluating individual code snippets, we believe they are relevant when comparing the quality of code produced by two different libraries as well. Both PyTorch and Tensorflow employs different design patterns and they are expected to be utilized in different ways [79], and it has been shown that different design patterns can have a significant impact on different code quality attributes, such as cohesion and complexity [74]. Therefore, we observed that there is a direct relationship between the design choice of libraries and the readability and understandability of code produced.

2.4 API Usability

To the best of our knowledge, there are no prior work that uses objective metrics to measure the usability of PyTorch and TensorFlow's APIs. Rama et al. [60] has proposed API usability metrics that are based on commonly held beliefs on poor API designs and anti-patterns. These include methods with a long list of parameters [44][53], not grouping conceptually similar API methods together [24][53], or just poor quality of API documentation [53]. It then proposes a variety of metrics that allows us to measure how well an API is designed and how well it avoids those anti-patterns.

However, those API metrics are designed for statically typed languages, such as Java. While both TensorFlow [12] and PyTorch [8] have Java APIs, they are not as popular, nor well supported, as their Python APIs [11]. It is because most researchers prefer Python as their language of choice [22]. This has become a challenge because Python is a dynamically typed language, making it impossible to determine the parameter type and return type of the methods before run-time [81]. To remedy this problem, we have scraped the official PyTorch [15] and TensorFlow [17] documentation to determine the expected parameter type and return type for each method. This will be further discussed in the methodology section.

Domain	Model Information		
	Model	**GitHub (TF)**	**GitHub (Torch)**
CV	ResNet50[40] VGG16[67] Incep-V3[69]	Benchmarks[70]	Examples[59]
NLP	GNMT[77]	DLExamples[56]	DLExamples[56]
	BERT[36]	BERT[38]	Transformer[42]
Speech	DeepS2[20]	Models[71]	deepspeech[64]

Table 1: Models in Different Domains and Repositories

3 BACKGROUND

3.1 Deep learning frameworks

Deep learning frameworks can be categorized into imperative and declarative frameworks. TensorFlow follows a declarative programming paradigm, whereas PyTorch adopts an imperative model. Given TensorFlow and PyTorch's prominence in academic circles, our comparison is restricted to these two frameworks.

TensorFlow, developed by Google, is a machine learning framework that employs a data flow graph to articulate computations. Within this graph, nodes represent mathematical operations, and edges depict the flow of data (Tensor in TensorFlow). Users utilize TensorFlow's API to construct models, which are subsequently translated into an internal computation graph during TensorFlow runtime. Throughout this process, TensorFlow can optimize the graph to enhance performance, employing techniques such as pruning, constant folding, and common subexpression elimination (CSE) to streamline computations[32]. However, the static nature of TensorFlow's computation graph poses challenges when dealing with recurrent neural networks (RNNs) that typically require a dynamic computation graph, making their handling more intricate.

PyTorch, an evolution of Torch and developed by Facebook, shares the dataflow paradigm with TensorFlow. However, it distinguishes itself by providing more flexible APIs capable of defining dynamic computation graphs. Embracing a "pythonic" coding style, PyTorch is characterized by its simplicity in learning and usage. Similar to TensorFlow, PyTorch employs Tensors as the core data abstraction, representing multi-dimensional arrays. Both PyTorch and TensorFlow facilitate users by offering automatic differentiation (backpropagation) for all operations on Tensors[32].

3.2 Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) serves as a hierarchical, tree-like representation of the syntactic structure inherent in programming code[55]. Comparable to a concise set of instructions for a task, an AST systematically dissects code into its elemental constituents such as variables, operations, and conditional statements. This hierarchical structure is akin to a tree, with the primary directive forming the root and subsequent components branching out in a structured manner. The AST functions as a fundamental intermediary in the process of code interpretation by a computer. It facilitates a systematic breakdown of intricate coding instructions, aiding in the analysis and subsequent execution of the code. In essence, the AST acts as a formalized blueprint that enhances computational comprehension and accuracy in executing complex programming directives.

3.3 Dynamically Typed Language

A dynamically typed language is a programming language where variable types are determined and checked during run-time rather than at compile-time. Compile-time refers to the period which the source code of a program is translated into machine code or byte-code by a compiler, while run-time refers to the period of time which the compiled machine code or byte-code is being executed [61].

In dynamically typed languages, the data type of a variable is not explicitly declared in the source code; instead, it is inferred or assigned when the program is executed. Python, which PyTorch [15] and TensorFlow [11] is targeting for, is a dynamically typed language [81]. Since the type is assigned only at run-time, it is impossible to determine the type of the variables without executing the code. While Python has support for type hinting, it does not enforce type hint annotations [18].

3.4 Web Scraping

Web scraping is a technique employed to extract and gather data from websites. It involves the automated retrieval of information from web pages by utilizing programs or scripts to navigate through the HTML structure of a website, locate relevant data, and extract it for analysis or storage. This process allows users to obtain large amounts of data from various online sources efficiently [76].

For our research, we have used web scraping techniques extensively on PyTorch [15] and TensorFlow's [11] official documentations. Since Python is dynamically typed, we will need to extract information within the documentation to determine the parameter type and its return type for a given API.

4 METHODOLOGY

4.1 Overview

In section 1, we outlined our primary objective of understanding the reasons behind researchers' preferences for PyTorch over TensorFlow in various scenarios. To achieve this, we will assess the ease of using APIs from both frameworks in both client code and documentation. Our approach involves employing diverse metrics for a quantitative comparison between the two frameworks. Additionally, we conduct a detailed analysis of bugs in the implementation of both frameworks, aiming to comprehend their implications in relation to community popularity. This section is divided into two main subsections: **API usability and Code Readability**, **API Density**, and **Issue Tracking**.

4.2 API Usability and Code Readability

In order to assess the smoothness of the API integration for each framework, we opted to evaluate and compare their APIs based on quantitative metrics. The methodology outlined in this subsection aims to provide insights into the quality of APIs, considering factors such as documentation clarity and the level of effort required by a programmer when writing code using either API framework.

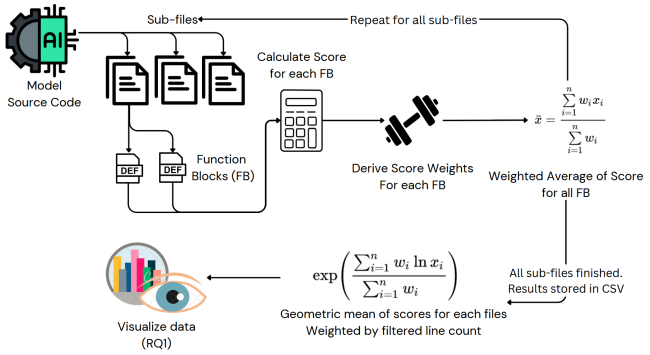

Figure 1: Client Code Metric Calculation Workflow

4.2.1 *Code Readability.*

Data Collection. We collected six diverse Deep Learning models, each implemented using both PyTorch and TensorFlow APIs. These models cover Computer Vision (3), Natural Language Processing (2), and Speech Recognition (1). Notably, each model has distinct implementations in both PyTorch and TensorFlow. To maintain research integrity, rigorous efforts were undertaken to source these models exclusively from reputable repositories affiliated with Google, Nvidia, and HuggingFace on the Git version control platform. For fairness, we excluded non-essential components, such as the training loop, dataset data loader, and extraneous functions, focusing solely on aspects directly related to the model implementation. Table 1 shows an overview of the models' name for each category identified.

Metrics Used. To assess the code readability and understandability, we implemented 3 key metrics following the steps outlined by Scalabrino et al. [62], Dantas et al [33]. and Campbell [26]. The details of the individual metrics we have chosen are as follows:

- Identifier Terms in Dictionary (**ITID**): Lawrie et al. [51] have shown that full-word identifiers ease source code comprehension. Therefore, Scalabrino et al. [62] has proposed the following metric to measure this property:

$$ITID(l) = \frac{|Terms(l) \cap Dictionary|}{|Terms(l)|} \quad (1)$$

  Where
  - l is the line of code
  - $Terms(l)$ are the terms within the line of code
  - $Dictionary$ is the set of words within the English dictionary

  However, there is a significant difference with our implementation. Instead of running this metric on all user code, we have selected only the portion of code that is provided by the API itself. For example, consider the following line of Python code:

```
tensor = torch.sparse_coo_tensor(indices,
         values,
         size=(2,2))
```

  The API has no control over the fact that the user has chosen the identifier "tensor" to receive the return of the API, nor that the parameters passed have names "indices" and "values".
  To remedy this issue, we have parsed the code into an Abstract Syntax Tree (AST), and we are only extracting method calls referring to the specific PyTorch / TensorFlow APIs. We then tokenize the method name into individual vocabularies, by using regular expressions and nltk's word tokenizer[6]. The English dictionary will be taken from nltk's wordnet lexical database, which is a standard dictionary already used for Natural Language Processing tasks [6].
- Textual Coherence (**TC**): Ujhazi et al. [72] has pointed out that the lack of cohesion between each class or block of code negatively impacts the code readability. Therefore, Scalabrino et al. [62] has proposed to compute the vocabulary overlap for each of the syntactic blocks of code, in order to measure TC. To implement this, we first parse the code into Abstract Syntax Trees (AST). We then identify syntactic blocks by checking for every control flow statements (e.g. 'if' statements). We then tokenize the code using regular expressions and nltk's word tokenizer [6]. The vocabulary overlap can be calculated as the set intersection between the set of tokens for each syntactic block.
- Cognitive Complexity (**CC**): Dantas et al. [33] has indicated that Cognitive Complexity (CC) [26] is a very good indicator of the understandability of a code snippet. Since the understandability score proposed by Dantas et al. is just a linear transformation of the CC score, we have decided to directly utilize the CC score for our metric.
  This result can be obtained by running SonarQube's Static Analyzer over the code for each PyTorch and Tensorflow models, with its score tabluated following the process described below.

Qualitative Analysis Methodology. Figure 1 has described the workflow for obtaining the Code Readability and Understandability Score. Each model is usually made up of multiple sub-files. For example, the PyTorch implementation of the GNMT model [56] contains multiple sub-files like "attention.py", "decoder.py", "encoder.py", etc. For each of these sub-files, we break down the Code into Function Blocks (FB). We then calculate the score for each of the FB, and tabulate the result.

We have noticed that none of these metrics have taken the length of the code, and the number of functions into account. It has been well known that longer code would result in higher code complexity [49]. Therefore, we will need to weight each of the FB's score with the following weight(w):

$$w = N \times \frac{l}{L} \quad (2)$$

Where N is the total number of functions inside the code file, l is the number of lines in a specific function, and L is the total number of lines inside the file.

This weight formula is then applied to all of the Function Blocks, and a weighted avearge for each of the model can be tabulated. Finally, to aggregate all of the results and generate a single number for each metric, we calculate the weight geometric mean for each

of the models. The weights for aggregating geometric mean will be the total number of lines needed to implement each of the model.

Refer to Figure 1 for the formulas of calculating weighted average and weighted geometric mean, where x_i is the unweighted score for each model / file, and w_i is the weight for the specific model / file.

4.2.2 *API Density.*

Data Collection. The methodology closely aligns with the data collection approach utilized for code readability. In this particular context, we employ the API density metric as a means of analyzing the client code.

Comparing Client Code using API density. We utilize API density metrics to assess the programming effort involved in constructing a model using either framework. This involves manually calculating the number of APIs within each of the six PyTorch models, and the same process is applied to each of the six TensorFlow models. The API density is subsequently calculated using the following equation:

$$APIDensity = \frac{NumberOFAPICalls}{NumberOfLines} \tag{3}$$

This calculation offers insights into the efficiency and complexity of model implementation, facilitating a clear comparison between using APIs for both frameworks.

4.2.3 *API Usability.*

Data Collection. As mentioned in the Related Works section, Python is a dynamically typed language [81]. In order to remedy this issue, we proposed to scrape the official documentation of PyTorch [15] and TensorFlow [11]. The documentation have documented the expected parameter types, and expected return types, for each of the APIs available.

To remove the need for building a web crawler, we have decided to build and host our own documentations locally. PyTorch have its website, containing all of its HTML documents, hosted on GitHub [16]. Meanwhile, TensorFlow has provided detailed guide on how to build documentations locally [14]. We followed the instructions provided and we are able to successfully obtain the documentations. Scraping PyTorch documentation is relatively straight forward, as the parameters, parameter types and return types are often at a fixed location, and exceptions are rare. However, scraping the types in TensorFlow has been a challenge, as it is described in English, rather than specifying which Python type it is.

To resolve this issue, we have extracted all the noun phrase inside the description of TensorFlow parameters, and build a translation table that translates the English noun to a Python type (e.g. "string" to "str"). This is done for all noun in the sentance, and we rank which one is more likely based on whether it is a TensorFlow type or a Python Build-in type.

We evaluated this method by randomly selecting 25 API Documentations, and compared its type identification result to our manual type identification. We discovered that it is able to achieve 89.43% accuracy when compared to our manual tagging result. Given our time and resource constraint, we believe this method does a satisfactory job in identifying the parameter and return types of TensorFlow APIs.

Metrics Used. To assess the API usability of both frameworks documentation, we implemented five key metrics adopted from Rama's et al. paper[60]. These metrics are employed to evaluate the API method declarations in light of widely accepted principles concerning factors that contribute to the complexity of API usage. We used these metrics on the data collected that discussed in the previous section.

- API Method Name Overload Index Metric (**AMNOI**): In Object-Oriented software systems, overloaded methods sharing the same name but with different return types can pose challenges for programmers [4]. The AMNOI metric gauges the extent to which various overload definitions for the same method name yield different return types. This is logically measured by determining the unique number of returns for each method divided by the total number of occurrences of that method in the API. The average is then calculated as shown below:

$$AMNOI = \frac{\sum_{i=1}^{n} 1 - \frac{\text{Unique Returns}_i - 1}{\text{Total Occurrences}_i - 1}}{n} \tag{4}$$

  where
  - n is the total number of distinct methods in the API.
  - Unique Returns$_i$ is the number of unique return types for the i-th method.
  - Total Occurrences$_i$ is the total number of occurrences of the i-th method in the API.
- API Parameter List Complexity Index Metric (**APXI**): This metric evaluates API method declarations, emphasizing a preference for a limited number of parameters (ideally 4 to 5) [31]. It gauges overall usability by considering the length of parameter sequences and their occurrence in consecutive runs of data objects of the same type. This metric is computed as the average of parameter length complexity (C_l) and parameter sequence complexity (C_s).

$$C_l = \frac{1}{M} \sum_{m=1}^{M} e^{min(0,(N_d - |A(m)|))} \tag{5}$$

$$C_s = \frac{1}{M} \sum_{m=1}^{M} h(m) \tag{6}$$

  where
  - N_d is the maximum number of parameters applicable. We considered it 5.
  - $|A(m)|$ represents the length of the parameter sequence in the method declaration m.
  - $h(m) = 1 - \frac{|Spt(m)|}{|A(m)|-1}$ if $|A(m)| > 1$. and $|Spt(m)|$ denotes the length of the sequence of parameters when the next parameter in the list is of the same type.

  Finally, the APXI is considered as the average as follows:

$$APXI = \frac{C_l + C_s}{2} \tag{7}$$

- API Method Name Confusion Index(**AMNCI**) Metric: during system evolution, developers often introduce new methods with names closely resembling existing ones although they somehow differ in the functionality [29]. This problem gives confusion to developers while using these methods.

More formally, a group of API methods is seen as confusing when the main names are identical, but this doesn't count overloaded versions of the same name. We manage this by ensuring that each method name appears only once in the group. We measure the confusion metric as follows:

$$AMNCI = 1 - \frac{|C|}{|M|} \tag{8}$$

where

- $|M|$ is the number of Methods.
- C is the list of confusing method names, where a name is confusing if there exist two members in M with identical canonical forms in C

AMNCI ranges between 0 and 1. If every method is confusing with some other, C equals M, and AMNCI is 0, but when all method names are distinct with no shared canonical forms,C is empty, and AMNCI is 1.

- API Method Grouping Index **(AMGI)** Metric: this metric addresses the challenge developers face in large libraries with numerous APIs, where organizing method declarations becomes pivotal for finding the most suitable functionality [25]. This metric gauges the extent to which similar APIs are grouped by functionality within the library.
  Before computation, common prefixes or suffixes are removed, such as "torch.nn". in PyTorch and TensorFlow. If the number of resulting fragments exceeds a threshold, denoted as theta, they are considered significant keywords. Then, we measure the grouping as follows: more mathematically, Let $S = \{s_j \mid j = 1..N\}$ be the set of significant keywords obtained as described above for a given API. For each keyword s_j, we sequentially substring match it with all the method names in the API in the order of declaration, considering each successful match as a 'run'. We construct a sequence of nonzero run lengths denoted as $L(s_j) = \{r_i \mid i = 1..R_j\}$, where R_j is the number of runs for the keyword s_j.
  Let r_i be the ith run-length value in $L(s_j)$. The total number of method declarations that match the keyword s_j is denoted as $O(s_j)$ and defined as:

$$O(s_j) = \sum_{i=1}^{R_j} r_i$$

We define the API Method Grouping Index (AMGI) for a given keyword s_j as:

$$AMGI(s_j) = 1 - \frac{(R_j - 1)}{O(s_j)}$$

For a keyword s_j, AMGI is equal to 1 when all method declarations containing the keyword exist in a single run in the API. At the other extreme, if method declarations with the keyword are completely scattered, AMGI will be 0.
The API-level AMGI metric is defined as the average of AMGI values for all keywords in S:

$$\text{AMGI} = \frac{1}{N} \sum_{j=1}^{N} AMGI(s_j)$$

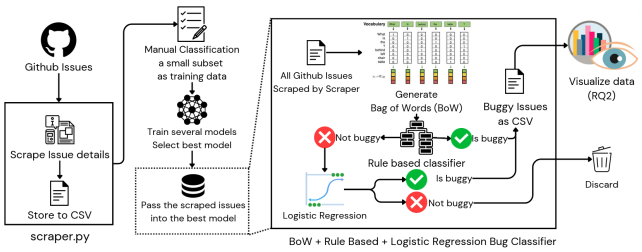

Figure 2: Github Issues Classification Workflow

The API-level metric AMGI normalizes the keyword-specific metric AMGI for a given API.

- API Documentation Index Metric(**DAI**): Detailed documentation is important for developers [13]. Therefore, the paper [60] has proposed to measure the quality of documentation as follows:

$$ADI = \frac{1}{|M|} \sum_{m \in M} \frac{L_d(m)}{d} \tag{9}$$

where

- d is the minimum acceptable method-related documentation.
- $L_d$is the number of words in the documntation.
- M is the total number of methods in the documentation.
  In our analysis, we made d to be **80** as the minimum acceptance.

Quantitative Analysis Methodology. The analysis method for API metrics is relatively straight forward. This is because each of the metrics above will generate a single percentage, and the numbers can be directly compared. A higher percentage indicates a better designed API in that specific category.

4.3 Issue Tracking

In Figure 2, we can see an overview of the issue tracking methodology. This subsection is explained in two main processes: data collection and qualitative analysis methodology.

4.3.1 *Data Collection.*

Issue Scraping. Our goal is to systematically analyze bug reports and ascertain the correlation between issues within the GitHub repositories of both PyTorch and TensorFlow. Initially, we utilized GitHub's API to automatically retrieve potential issues for each framework. Pull requests were excluded from the data collection process, as we are unnecessary for our research objectives. Given our focus on comprehending the evolution of bug reports over time, we extracted information from both open and closed issues. In total, we gathered 37,753 PyTorch issues and 38,288 TensorFlow issues, encompassing details such as issue titles, bodies, creation timestamps, closure timestamps (if applicable), and associated tags. The rationale behind scraping these specific features is outlined as follows:

- the inclusion of **the issue title, issue body, and tags** proves instrumental in the classification of the nature of the

issue, determining whether it is indeed a bug. Further details on this aspect are provided in the subsequent subsection.
- the timestamps indicating the time of **issue creation and closure** are imperative for gauging the overall community activity in reporting and resolving issues.

Manual Classification. Identifying the buggy issues from 37753 PyTorch issues and 38,288 TensorFlow issues using manual classification is not feasible. In accordance with the methodology proposed by Lopez et al. [34], a simple but powerful approach was adopted, subjecting a subset of issues from both PyTorch and TensorFlow to manual classification. This curated subset was then utilized to train a machine learning model, as explained in the subsequent subsection. Specifically, 194 PyTorch issues and 245 TensorFlow issues were meticulously classified. Fortunately, We reached out to the authors of [41] and got confirmation on 194 PyTorch buggy issues. Similarly, we contacted the authors of [46] and confirmed 97 TensorFlow buggy issues. Combining these, we had a set of **388** PyTorch issues and **342** TensorFlow issues. Notably, the data was balanced to ensure that exactly half of the issues in each category were identified as non-buggy. The process of classifying the issues was done in two-stages. The issue classification process occurred in two stages, with the initial identification performed by the first author and subsequent confirmation of labels by the second author. Finally, we negotiated the ones we did not agree on. The procedure for selecting issues for manual classification was implemented using a random sampling methodology.

Bug Classification. In this section, we outline how we identified whether both closed and open issues are buggy or not. We utilized a machine learning model trained using manually classified issues. To ensure robustness, we experimented with different methods and selected the most effective based on precision and recall metrics. Upon analysis, we observed format differences between PyTorch and TensorFlow issues, leading us to employ separate models for each framework to avoid potential noise or biases. As outlined in the issue scraping subsection, input data for the model includes issue titles, bodies, and tags, with two possible combinations: combining the issue title with both the body and tags, or with tags alone. Next, we mapped both combinations with the semantic space using four methodologies: word embedding utilizing Word2Vec[54], transformer embedding employing bert embedding [36], bag of words with frequency approach[78], and bag of words using TF-IDF[78]. For each technique, various Machine Learning models were explored, including logistic regression, Naive Bayes, Random Forests, Decision Trees, and a basic Neural Network. Given the limited data, cross-validation was employed. The outcomes revealed that logistic regression, coupled with a bag of words using the frequency approach and issue title with tags only as input data, yielded the most favorable results for both manually labeled TensorFlow and PyTorch issues. Specifically, we achieved **94.6%** precision and **94.8%** recall for PyTorch machine learning model, while TensorFlow machine learning model exhibited **92.4%** precision and **92.3%** recall. The reason behind obtaining dire results with a combination

For the purpose of enhancing robustness, our bug classification methodology is structured into two stages. Initially, we determine the bugginess of PyTorch issues exclusively based on input from issue titles and tags. First, a word checker is incorporated to identify

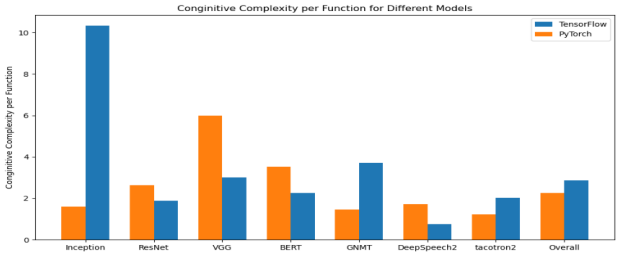

Figure 3: Cognitive Complexity for 7 models: The lower is the better

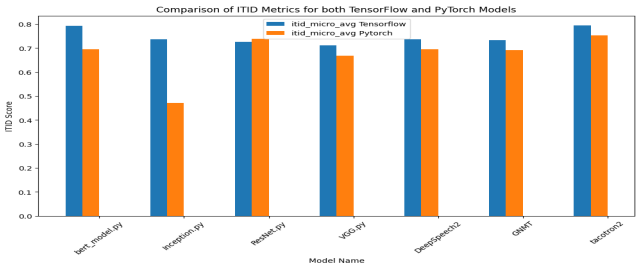

Figure 4: ITID results for each model

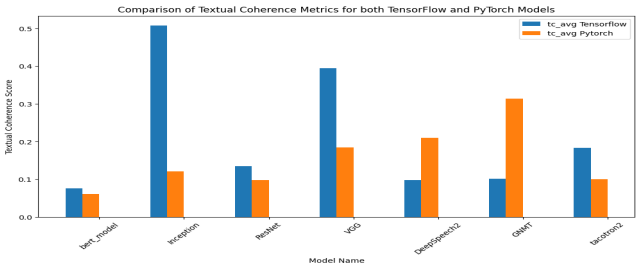

Figure 5: textual coherence results for each model

prevalent terms, such as "crash" and "segmentation fault," within both the titles and tags. If any of these terms are present, the issue is classified as buggy. If not, the second-stage is applied. The unconsidered buggy from first stage will be classified using PyTorch Logistic regression model. This identical process is applied to TensorFlow issues.

4.3.2 ***Qualitative Analysis Methodology.*** Upon categorizing the issues for each framework, we were confronted with two pivotal inquiries essential for addressing RQ2. Firstly, in what manner do bugs evolve over time in both frameworks? Secondly, what is the temporal efficiency in terms of response time for issue resolution in both frameworks? Given the data, we produced multiple graphs that will be explained further in the results section.

5 RESULTS

In this section, we present our findings for each research question. To assess usability, we applied 10 metrics across various dimensions to understand how seamlessly APIs are utilized within each framework. More specifically, 5 metrics discussed in the methodology

section measure the code readability and API density in client code, while the other 5 metrics measure the documentation of both frameworks. Concerning the evolution of bugs, we conducted a statistical analysis to explain why PyTorch has become more prevalent, relying on 38,335 PyTorch issues and 38,421 TensorFlow issues. It is important to note that pull requests were not taken into account in our analysis. This section serves to report both the results of the research questions and their corresponding implications.

5.1 RQ1: Which framework facilitates seamless utilization of API?

5.1.1 Code Readability. As outlined in the methodology, this section assesses code readability and understandability when implementing deep learning (DL) models with Framework APIs. Figure 3 indicates that utilizing TensorFlow APIs leads to more complex code compared to PyTorch API on average overall because the lower should be better. However, a closer examination, as detailed in the methodology, reveals that cognitive complexity, determined by if-branch conditions, is subjective and dependent on the programmer, even when leveraging client codes from top companies.

To delve into code readability, Figure 4 illustrates instances where implementing DL models with TensorFlow APIs can be advantageous, such as the BERT Model with an 80% ITId score compared to 70% if used PyTorch APIs. Conversely, there are cases, like the ResNet model, where using PyTorch APIs yields better results (70% for PyTorch vs. 68% for TensorFlow). However, the majority of differences in percentage are statistically insignificant, resulting in inconclusive findings. This ambiguity is exacerbated when examining textual coherence results in Figure 5. For example, the Inception model implemented with TensorFlow exhibits superior textual coherence, while the GNMT model implemented with PyTorch attains a significantly higher score of 30% compared to TensorFlow's 10%, further contributing to inconclusive results.

Metric		Inception	ResNet	VGG16	BERT	GNMT	DSpeech
Density (%)	PyTorch	35.3	32.2	58.3	59.1	46.1	33.4
	TensorFlow	45.1	36.8	49.7	74.2	47.2	36.7
Unique API (Automated)	PyTorch	14	8	8	16	11	11
	TensorFlow	9	10	9	22	41	10

Table 2: Density and Unique API for Models

5.1.2 API density. Table 2 illustrates the API density across various model implementations using both PyTorch and TensorFlow. A noticeable trend emerges, indicating that, on average, PyTorch models exhibit lower API density in comparison to TensorFlow models, except in the case of VGG16. This observation prompted us to delve deeper into the discrepancy, especially since we conducted this analysis across six distinct models.

To shed light on this phenomenon, consider the example of the BERT Model Figure 6. It becomes evident that TensorFlow requires a higher number of API calls to perform a seemingly straightforward task of multiplying two weights to implement a linear layer. In contrast, PyTorch simplifies this process by providing a more concise solution through the use of nn.linear. This discrepancy highlights a significant advantage in terms of code simplicity and efficiency for PyTorch, contributing to the observed differences in API density.

```
# TensorFlow Implementation
def call(self, hidden_states: tf.Tensor) -> tf.Tensor:
    hidden_states = self.transform(hidden_states=hidden_states)
    seq_length = shape_list(hidden_states)[1]
    hidden_states = tf.reshape(tensor=hidden_states, shape=[-1,
    ↪  self.hidden_size])
    hidden_states = tf.matmul(a=hidden_states,
    ↪  b=self.input_embeddings.weight, transpose_b=True)
    hidden_states = tf.reshape(tensor=hidden_states, shape=[-1,
    ↪  seq_length, self.config.vocab_size])
    hidden_states = tf.nn.bias_add(value=hidden_states,
    ↪  bias=self.bias)
    return hidden_states
# ========================================================== #
# PyTorch Implementation
def forward(self, hidden_states):
    hidden_states = self.transform(hidden_states)  # Just
    ↪  nn.transform(config)
    hidden_states = nn.Linear(hidden_states)  # nn.Linear
    return hidden_states
```

Figure 6: forward function for BertLMPredictionHead

This argument gains further support when examining the unique number of APIs for each model. With the exception of Inception, the other five models require significantly fewer PyTorch API calls for implementation. To illustrate, consider GNMT, which demands 41 unique TensorFlow API calls compared to only 11 unique PyTorch API calls. This observation reinforces the notion that PyTorch tends to achieve similar functionalities with a more concise and efficient API usage, contributing to the overall differences in API density across models.

Metric	AMGI	AMNCI	AMONI	APXI	DAI	Number of APIs
PyTorch	74.68%	87.06%	99.88%	71.43%	90.48%	1748
TensorFlow	75.99%	83.28%	99.38%	69.94%	42.57%	3232

Table 3: API Metric Scores for PyTorch and TensorFlow

5.1.3 API Usability. Table 3 provides an overview of PyTorch and TensorFlow's score for each of the metrics described in section 4.3.1. From the result, we can see that apart from AMGI, PyTorch has outscored TensorFlow in every other metrics we have proposed. In particular, PyTorch is able to outscore TensorFlow in AMNCI by 3.78%, APXI by 1.49%, and most significantly, DAI by 47.91%. The DAI result may seems outrageous, given the huge difference in the number in favor of PyTorch. However, TensorFlow's documentation already has a very poor reputation among Machine Learning developers. One reported that the official TensorFlow documentation is very lacking, and very often it is required to rely on third party tutorials and repos to understand certain APIs [43]. It also claims that TensorFlow's API Documentation is especially lacking for newer APIs.

There are also reports saying that TensorFlow's APIs are difficult to remember [73]. While both PyTorch nor TensorFlow scored high in the corresponding metric (AMGI) that measures API consistency, PyTorch has vastly fewer APIs. TensorFlow has 84.9% more APIs than PyTorch, meaning that users are often required to remember more APIs in order to achieve the same functionality. This is further

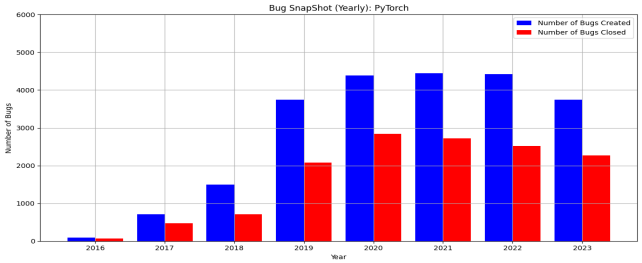

Figure 7: Number of bugs reported per year (PyTorch)

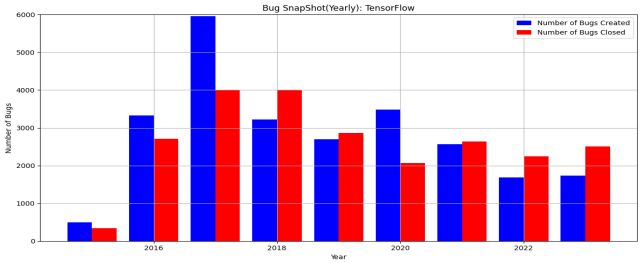

Figure 8: Number of bugs reported per year (TensorFlow)

support in our result of API Density (Table 2)), as we can see that PyTorch needs a lot less unique API calls, and less API calls in general, for most of the model.

> *Answer to RQ1*: While the results of Code Readability is inconclusive, PyTorch is better in Code Understandability. Furthermore, PyTorch has generally lower API Density, more usable APIs, and better Documentation Quality.

5.2 RQ2: How does the prevalence of bugs in each framework influence their respective popularity over time?

Figure 7 and Figure 8 show the number of bugs opened and closed over time for PyTorch and TensorFlow respectively. We can observe that their distribution is vastly different. PyTorch has a steady increase in the number of issues created over time, while TensorFlow has a rapid rise in the number of issues from 2015 to 2017 and then a sudden drop from 2017 to 2018, and it has leveled off since then.

First, let's look at how TensorFlow's bugs over time relate to the popularity of the framework. 2015 to 2016 is a significant growth period in application of artificial intelligence (AI) [23][28], and Python has become the language of choice for a lot of researchers in AI [22]. Therefore, it is not surprising that there is a steady growth in the number of bugs reported over time, as more and more people start to use this library. Therefore, more and more bugs may be discovered because there are more people testing the features of the library.

That being said, the number of bugs reported is much higher than expected. In Figure 8, we can see that there are a total of nearly 6000 bugs reported in the year 2017 alone. Meanwhile, looking at

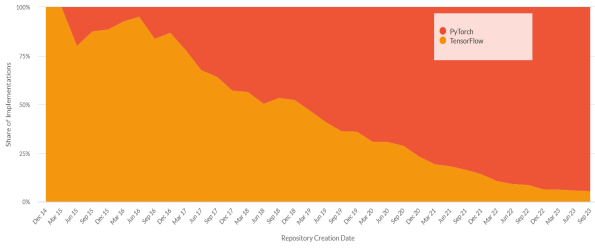

Figure 9: Percentage of papers published using PyTorch or TensorFlow, from 2015 to 2023 [7]

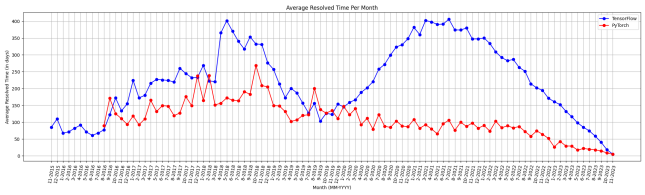

Figure 10: Average Issue Resolved Time of TensorFlow and PyTorch

PyTorch (Figure 7), It only received at most around 4500 bug reports during its worst years. Given that PyTorch is much more widely used than TensorFlow as seen in Figure 9, this is unusual.

We hypothesize the cause for this is because TensorFlow, for the versions released between 2017 and 2019, is simply difficult to use. This is known as TensorFlow 1.x. There is multiple evidence:

- Poor API Design: TensorFlow 1 has a poorly designed API interface. This is evident by TensorFlow's migration documentation [10]. In the document, it has pointed out a lot of shortcomings of TensorFlow 1's APIs. For instance, TensorFlow 1 relied heavily on global variables, and it requires the developer to manually stitch together an AST in order to build the compute graph. This requires developers having an in-depth understanding of the working principle of the API, and it is not a good API design pattern [53].
- Documentation Quality: TensorFlow's documentation quality is not as good as PyTorch. It is evident from both the API Documentation Index Metric (DAI) result we have collected (Table 3), as well as opinions from AI developers in the industry [43][50][73] . This may lead to API misuse, which may cause software crashes and unexpected behaviors [68]. Therefore, users are more likely to file a bug report when using TensorFlow instead of PyTorch.
- Slow Response Time: In Figure 10, we can see that from the period between October 2016 and June 2019, the TensorFlow community has consistently required more time to resolve an issue than the PyTorch community. This can suggest that the community for TensorFlow is not as active as PyTorch, or it could simply be because the number of bugs is higher in TensorFlow than PyTorch (Figure 8). Nonetheless, This means that users will have to wait for a longer period of time in order to get their bug resolved by the developers, as much as 60%.

However, we then see a sharp decline in the number of bugs reported from 2017 to 2019, and it keeps decreasing from 2020 onwards. There is only a slight increase from 2019 to 2020, due to the release of TensorFlow 2 [1]. Meanwhile, we can also see that PyTorch gained significant traction. Between March 2017 and September 2019, the number of AI related papers published using PyTorch as the library of choice has increased from under 15% to over 70% in the span of 2 years (Figure 9).

The drop in the number of bugs reported in TensorFlow is not a sign of drastically improved API usability or API quality. This is because there are no major releases between 2017 and 2018. The Closest major release, TensorFlow 2, is not released until 2019 [1]. We interpret this as a sign of shifting interest from TensorFlow to PyTorch. This is evident from Figure 9, and we can see that the number of papers published using TensorFlow has significantly decreased starting from March 2017. If we look at the number of bugs reported in Figure 8, there are around 6000 bugs reported for TensorFlow, while there are only around 700 bugs reported for PyTorch (Figure 7). Hence, evidently, researchers are looking into PyTorch as an alternative to TensorFlow.

This is further supported when we look at the issue resolve time (Figure 10.) While there is a decline between 2018 and 2019, due to the decrease in the number of users hence less bug reports are filed, the response time has increased steadily between 2019 and 2021. While the trend is decreasing as of 2023, PyTorch still has a much lower average response time in general. This suggests that the TensorFlow community is not as active compared to PyTorch, due to its decreasing popularity. We can also see that (Figure 9), despite the release of TensorFlow 2 in 2019, researchers do not see a motivation to switch back to TensorFlow.

> *Answer to RQ2*: The unusually high amount of bugs in TensorFlow, during 2016 - 2017, has pushed researchers to the alternative framework: PyTorch. This is further compounded by TensorFlow's slow issue resolve time and less active community.

6 THREADS TO VALIDITY

6.1 Internal Validity

Any tools we have created could be inaccurate, or we could be overly optimistic of the actual performance of such tools. However, given our limited time and resources, as well as the lack of alternatives, there are very few options available. For all of the metrics, there are no code implementations publicly available. There is also no public dataset available that documents the parameter and return types of PyTorch / TensorFlow APIs, therefore some form of scraping is always necessary.

To ensure correctness of our tools, we have both validated the tools with validation data, and we would manually review the data generated. All the tools we produced, such as the web scraping tool and the type extraction tool, are the result of many iterations of method refinement and manual validation of results. While there may be some minor inaccuracies in the data generated, they account for a very small percentage based on our validation results.

[1] https://github.com/tensorflow/tensorflow/releases/tag/v2.0.0

6.2 External Validity

Our research relies on external, public client code. Therefore, there is a possibility that the code is not the highest possible standard, or the most efficient usage of APIs. To reduce such possibility, we have carefully selected the relevant repository, and always relied on official implementation from the library developers themselves. Since the library developers should be the most familiar with their own libraries, the chance of API misuse and inefficient code should be minimal.

We are also relying on the correctness of the official documentation of PyTorch and TensorFlow. While some have reported that TensorFlow documentation provides minimal examples for newly released APIs, they are rare occurrences and only limited to newly released APIs [43]. Meanwhile, PyTorch's documentation has been regarded as very good and up to date, so the risk of incorrect documentation should be minimal[63][50].

7 CONCLUSIONS AND FUTURE WORK

In this study, we investigated the preference for PyTorch over TensorFlow in implementing novel models. Our comprehensive analysis, both quantitative and qualitative, focused on usability functionality and bug evolution. Our findings indicate that (1) implementing Deep Learning models using PyTorch APIs requires less effort compared to TensorFlow, and (2) the PyTorch community has significantly improved, resolving bugs more promptly than the TensorFlow community. This study provides valuable insights for developers creating frameworks, emphasizing key features crucial for researchers. Future research should delve deeper into bug evolution by categorizing types and patterns, and exploring the usability of API calls in Deep Learning models.

8 REPLICATION PACKAGE

We have made our codes, data, and detailed replication steps publicly available in our repository. You can access our work through the following link: https://github.com/mohamedayman15069/A-Comparative-Study-between-TensorFlow-and-PyTorch.

REFERENCES

[1] [n. d.]. *Django Documentation.* https://docs.djangoproject.com/en/4.2/ Accessed: December 6, 2023.
[2] [n. d.]. *Flask Documentation.* https://flask.palletsprojects.com/en/3.0.x/ Accessed: December 6, 2023.
[3] [n. d.]. *Google Trends Data for TensorFlow vs. PyTorch.* https://trends.google.com/trends/explore?q=TensorFlow,PyTorch Accessed: December 6, 2023.
[4] [n. d.]. *Learning the java language - overloading methods.* https://docs.oracle.com/en/ Accessed: December 6, 2023.
[5] [n. d.]. *Matplotlib Documentation.* https://matplotlib.org/stable/index.html Accessed: December 6, 2023.
[6] [n. d.]. *Natural Language Toolkit (NLTK) Documentation.* https://www.nltk.org/index.html Accessed: December 6, 2023.
[7] [n. d.]. *Papers with Code Trends: PyTorch vs. TensorFlow.* https://paperswithcode.com/trends Accessed: December 6, 2023.
[8] [n. d.]. *PyTorch 1.9.0 Javadoc.* https://pytorch.org/javadoc/1.9.0/ Accessed: December 6, 2023.
[9] [n. d.]. *Seaborn API Reference.* https://seaborn.pydata.org/api.html Accessed: December 6, 2023.
[10] [n. d.]. *TensorFlow 1.x vs TensorFlow 2 - Behaviors and APIs.* https://www.tensorflow.org/guide/migrate/tf1_vs_tf2 Accessed: December 6, 2023.

[11] [n. d.]. *Tensorflow API Documentation.* https://www.tensorflow.org/api_docs Accessed: December 6, 2023.

[12] [n. d.]. *TensorFlow for Java.* https://www.tensorflow.org/jvm Accessed: December 6, 2023.

[13] [n. d.]. Why documentation is important in software development. https://www.linkedin.com/pulse/why-documentation-important-software-development-alexander-ryan/ Accessed: December 6, 2023.

[14] 2023. *Contribute to the TensorFlow documentation.* https://www.tensorflow.org/community/contribute/docs Accessed: December 6, 2023.

[15] 2023. *Pytorch API Documentation Github Repository.* https://github.com/pytorch/pytorch.github.io Accessed: December 6, 2023.

[16] 2023. *pytorch.github.io.* https://github.com/pytorch/pytorch.github.io Accessed: December 6, 2023.

[17] 2023. *Tensorflow API Documentation Github Repository.* https://github.com/tensorflow/docs Accessed: December 6, 2023.

[18] 2023. *typing - Support for type hints.* https://docs.python.org/3/library/typing.html Acccessed: December 6, 2023.

[19] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[20] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*. PMLR, 173–182.

[21] AssemblyAI. [n. d.]. *PyTorch vs. TensorFlow in 2023.* https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/ Accessed: December 6, 2023.

[22] Manasi Bandichode. 2023. *Programming Language Industry in 2023: A Comprehensive Overview.* https://www.linkedin.com/pulse/programming-language-industry-2023-comprehensive-manasi-bandichode/ Accessed: December 6, 2023.

[23] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). arXiv:1604.07316 http://arxiv.org/abs/1604.07316

[24] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[25] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1589–1598.

[26] G. Ann Campbell. 2018. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18)*. Association for Computing Machinery, New York, NY, USA, 57–58. https://doi.org/10.1145/3194164.3194186

[27] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (sep 2023), 31 pages. https://doi.org/10.1145/3587155

[28] Travers Ching, Daniel S Himmelstein, Brett K Beaulieu-Jones, Alexandr A Kalinin, Brian T Do, Gregory P Way, Enrico Ferrero, Paul-Michael Agapow, Michael Zietz, Michael M Hoffman, et al. 2018. Opportunities and obstacles for deep learning in biology and medicine. *Journal of The Royal Society Interface* 15, 141 (2018), 20170387.

[29] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. 2000. Alice: Lessons Learned from Building a 3D System for Novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 486–493.

[30] NVIDIA Corporation. 2023. NVIDIA Deep Learning Examples. https://github.com/NVIDIA/DeepLearningExamples. Accessed: December 6, 2023.

[31] Nelson Cowan. 2001. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences* 24, 1 (2001), 87–114. https://doi.org/10.1017/S0140525X01003922

[32] Hulin Dai, Xuan Peng, Xuanhua Shi, Ligang He, Qian Xiong, and Hai Jin. 2022. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment. *Science China Information Sciences* 65 (2022), 1–17.

[33] Carlos Dantas and Marcelo Maia. 2021. Readability and Understandability Scores for Snippet Assessment: an Exploratory Study. *Anais do Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, 46–50. https://doi.org/10.5753/vem.2021.17217

[34] Fernando López de la Mora and Sarah Nadi. 2018. An Empirical Study of Metric-Based Comparisons of Software Libraries. 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, 22–31. https://doi.org/10.1145/3273934.3273937

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://github.com/google-research/bert. Accessed: December 6, 2023.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[37] Devndra Ghimire. 2020. Comparative study on Python web frameworks: Flask and Django. (2020).

[38] Google Research Authors. Year. BERT: Pre-trained models and downstream applications. https://github.com/google-research/bert.git. Accessed: Date.

[39] Abdul Hafeez and Ali Sial. 2021. Comparative Analysis of Data Visualization Libraries Matplotlib and Seaborn in Python [HEC Y Cat]. *International Journal of Advanced Trends in Computer Science and Engineering* 10 (02 2021), 2770–281. https://doi.org/10.30534/ijatcse/2021/391012021

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[41] Sharon Chee Yin Ho, Vahid Majdinasab, Mohayeminul Islam, Diego Elias Costa, Emad Shihab, Foutse Khomh, Sarah Nadi, and Muhammad Raza. 2023. An Empirical Study on Bugs Inside PyTorch: A Replication Study. arXiv:2307.13777 [cs.SE]

[42] Hugging Face Authors. Year. Transformers: State-of-the-art Natural Language Processing for TensorFlow 2.0 and PyTorch. https://github.com/huggingface/transformers. Accessed: Date.

[43] indicodata.ai. 2016. The Good, Bad, & Ugly of TensorFlow. (2016). https://indicodata.ai/blog/the-good-bad-ugly-of-tensorflow/

[44] Bloch J. 2001. *Effective Java: Programming Language Guide, The Java Series*. Addison-Wesley: New York.

[45] Arpan Jain, Ammar Ahmad Awan, Quentin Anthony, Hari Subramoni, and Dhableswar K. DK Panda. 2019. Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. https://doi.org/10.1109/CLUSTER.2019.8891042

[46] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 604–620.

[47] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[48] Valantis K. 2023. *Battle of The Giants: TensorFlow vs PyTorch 2023.* https://medium.com/@valkont/battle-of-the-giants-tensorflow-vs-pytorch-2023-fd8274210a38 Accessed: December 6, 2023.

[49] Naveen Kumar. 2021. *What is Code Complexity? Why it Matters and How to Measure it.* https://www.hatica.io/blog/code-complexity/ Accessed: December 6, 2023.

[50] Vihar Kurama. 2020. PyTorch vs. TensorFlow: Key Differences to Know for Deep Learning. (2020). https://builtin.com/data-science/pytorch-vs-tensorflow

[51] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *ISSE* 3 (11 2007), 303–318. https://doi.org/10.1007/s11334-007-0031-2

[52] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 34, 2 (2008), 287–300. https://doi.org/10.1109/TSE.2007.70768

[53] ZeroC Michi Henning. 2007. API: Design Matters. (2007). https://dl.acm.org/doi/fullHtml/10.1145/1255421.1255422

[54] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]

[55] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*. 1–5.

[56] NVIDIA Corporation. Year. NVIDIA Deep Learning Examples. https://github.com/NVIDIA/DeepLearningExamples.git. Accessed: Date.

[57] Keiron O'Shea and Ryan Nash. 2015. An Introduction to Convolutional Neural Networks. *CoRR* abs/1511.08458 (2015). arXiv:1511.08458 http://arxiv.org/abs/1511.08458

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[59] PyTorch Developers. Year. PyTorch Examples. https://github.com/pytorch/examples.git. Accessed: Date.

[60] Girish Maskeri Rama and Avinash Kak. 2015. Some structural measures of API usability. *Software: Practice and Experience* 45, 1 (2015), 75–110. https://doi.org/10.1002/spe.2215 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2215

[61] Sandip Roy. 2023. *Runtime vs. Compile Time.* https://www.baeldung.com/cs/runtime-vs-compile-time Accessed: December 6, 2023.

[62] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018), e1958. https://doi.org/10.1002/smr.1958 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1958 e1958 smr.1958.

[63] Toward Data Science. 2020. Reasons to Choose PyTorch for Deep Learning. (2020). https://towardsdatascience.com/reasons-to-choose-pytorch-for-deep-learning-c087e031eaca

[64] Sean Naren. Year. DeepSpeech PyTorch. https://github.com/SeanNaren/deepspeech.pytorch. Accessed: Date.

[65] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2135–2135.

[66] Alex Sherstinsky. 2018. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *CoRR* abs/1808.03314 (2018). arXiv:1808.03314 http://arxiv.org/abs/1808.03314

[67] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[68] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API-Misuse Detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 265–275. https://doi.org/10.1109/MSR.2019.00053

[69] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[70] TensorFlow Authors. Year. TensorFlow Benchmarks. https://github.com/tensorflow/benchmarks.git. Accessed: Date.

[71] TensorFlow Authors. Year. TensorFlow Models. https://github.com/tensorflow/models.git. Accessed: Date.

[72] Bela Ujhazi, Rudolf Ferenc, Denys Poshyvanyk, and Tibor Gyimothy. 2010. New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems. *Proceedings - 10th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010*, 33 – 42. https://doi.org/10.1109/SCAM.2010.14

[73] Venkatesh Wadawadagi. 2023. *TensorFlow vs PyTorch: Deep Learning Frameworks.* https://www.knowledgehut.com/blog/data-science/pytorch-vs-tensorflow#advantages-and-disadvantages-of%C2%A0tensorflow%C2%A0 Accessed: December 6, 2023.

[74] Fadi Wedyan and Somia Abufakher. 2020. Impact of design patterns on software quality: a systematic literature review. *IET Software* 14, 1 (2020), 1–17. https://doi.org/10.1049/iet-sen.2018.5446 arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2018.5446

[75] Thomas Wolf, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, et al. 2023. Transformers: State-of-the-art Natural Language Processing for TensorFlow 2.0 and PyTorch. https://github.com/huggingface/transformers. Accessed: December 6, 2023.

[76] Songhao Wu. 2020. *Web Scraping Basics.* https://towardsdatascience.com/web-scraping-basics-82f8b5acd45c Accessed: December 6, 2023.

[77] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[78] Dongyang Yan, Keping Li, Shuang Gu, and Liu Yang. 2020. Network-Based Bag-of-Words Model for Text Classification. *IEEE Access* 8 (2020), 82641–82652. https://doi.org/10.1109/ACCESS.2020.2991074

[79] Eugene Yan. 2022. *Design Patterns in Machine Learning Code and Systems.* https://eugeneyan.com/writing/design-patterns/ Accessed: December 6, 2023.

[80] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph Neural Networks: A Review of Methods and Applications. *CoRR* abs/1812.08434 (2018). arXiv:1812.08434 http://arxiv.org/abs/1812.08434

[81] Sadia Zubair. [n. d.]. *Is Python a dynamically typed language.* https://www.educative.io/answers/is-python-a-dynamically-typed-language Accessed: December 6, 2023.