Deep Code Curator – code2graph Part-I

Shih-Yuan Yu
Ahmet Salih Aksakal
Sujit Rokka Chhetri
Mohammad Abdullah Al Faruque

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{shihyuay, aaksakal, schhetri, alfaruqu}@uci.edu

CECS Technical Report TR# 19-01
April 21, 2019

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00111990010
Contents

1 Introduction ........................................................................................................................................... 3

2 Proposed Methodology ..................................................................................................................... 5
  2.1 Preprocessing ................................................................................................................................... 5
    2.1.1 Python Version Check ............................................................................................................. 6
    2.1.2 Resolving requirements .......................................................................................................... 6
    2.1.3 Code Injection .......................................................................................................................... 6
    2.1.4 Challenges on Automating the Re-creation of Environment ...................................................... 8
  2.2 Computational Graph-based Approach ........................................................................................... 8
    2.2.1 Dataflow Extraction: ................................................................................................................. 8
    2.2.2 Protocol Buffer Parser ............................................................................................................. 9
    2.2.3 Converting Json to RDF Graph: ................................................................................................. 10
  2.3 Ontology and Vocabulary Extraction ............................................................................................. 14
  2.4 Light-Weight Method ..................................................................................................................... 15
    2.4.1 AST Extraction .......................................................................................................................... 15
    2.4.2 Call Graph Extraction .............................................................................................................. 15
    2.4.3 Graph Parsing ............................................................................................................................ 17
    2.4.4 Sequence Extraction ................................................................................................................. 17

3 Code Implementation Detail ................................................................................................................. 17
  3.1 Computational graph-based Approach ............................................................................................ 17
  3.2 Lightweight Approach ..................................................................................................................... 19

4 Results .................................................................................................................................................... 20
  4.1 Computational Graph Extraction .................................................................................................... 20
    4.1.1 Test on simple graph ................................................................................................................. 20
    4.1.2 Test on real scientific paper’s code repository .......................................................................... 21
  4.2 Lightweight Graph Extraction ......................................................................................................... 22
    4.2.1 Test on simple graph ................................................................................................................. 22
    4.2.2 Test on complicated graph ....................................................................................................... 23
  4.3 Jupyter Notebook Example ............................................................................................................. 25
4.3.1 Computation graph-based graph extraction .........................................................25
4.3.2 Lightweight-based method .........................................................................................26

5 Future Work .............................................................................................................................28
5.1 Filtering Architecture RDF from unnecessary data .........................................................28
5.2 Generating Code from RDF graph .....................................................................................28
  5.2.1 Rule-based preprocessing of the RDF graph: .................................................................29
  5.2.2 Adding context/attribute to the RDF graph: .................................................................29
  5.2.3 RDF graph to tf.Graph(): ...........................................................................................30
  5.2.4 Evaluation of the Graph2Code: ....................................................................................30
  5.2.5 Converting tf.Graph() structure to code: .................................................................30
5.3 Knowledge Graph Embedding ...........................................................................................31
5.4 Similarity Measure and Novelty Measure in Embedding Space ......................................31

6 Discussion ...............................................................................................................................32

7 References ..............................................................................................................................32
1 Introduction

Deep Learning (DL) is a fast-growing field with thousands of researchers working on new models and algorithms every day. Usually, deep learning models require the experts to spend their time reading through the latest publications, making sense of the claims and the results, and using their experience to create links between the body of knowledge to validate new ideas. However, as deep learning algorithms and implementations across various fields grow at an explosive rate, keeping up with all the latest publications and source code becomes a challenge for researchers and practitioners who hope to contribute their work to the community of deep learning.

Therefore, the goal of the project - Deep Code Curator (DCC) - aims to tackle with this issue by utilizing the information from scientific publications and the corresponding source code related to deep learning architectures and methodologies. The DCC will analyze the scientific papers and represent them in a universal representation that can dramatically decrease the time, effort and resources spent curating deep learning literature and algorithms. In DCC, the modalities of a scientific paper, which contain text, images, equations, and code, are processed via the support from modules text2graph, image2graph, and code2graph separately. The goal of these modules is to create the Resource Description Framework (RDF) knowledge graphs that can be later processed and merged into a super-graph. This super-graph serves as the architectural representation of a scientific paper and can be used to explore and compare the papers across various fields that share similar deep learning architectures. Besides, the module graph2code even aims to process this super-graph and convert it to actual code implementation, which can help deep learning experts to evaluate their ideas more efficiently.

This technical report will focus on introducing the pipeline module code2graph in DCC. The main aim of the module code2graph is to extract the architectural information from the existing code repositories corresponding to the scientific papers in the domain of deep learning. To be more specific, knowledge graphs in the form of RDF graphs are extracted from the deep learning programs using Python machine learning frameworks such as TensorFlow and Keras. The main aim in extracting the knowledge graphs from code is two-fold. First, it will serve to complete the information extracted from the text and the images of the scientific graph. This aggregated and fused graph will serve as the super graph from which we aim to perform novelty and similarity detection among other scientific papers. The graph also helps to train stochastic models to aid in completing the inferred code graph utilizing the traditional rule-based inference methodologies utilized in extracting information from the knowledge graphs. Moreover, models can be created to convert the inferred knowledge graphs back to code template to aid in reproducibility of the scientific papers.

There has been proliferation of works that analyze the code to extract meaningful abstractions, summaries [1], conventions [2], patterns [3], algorithms [4] etc. However, the works that are closest in converting the code data structure to meaningful graph or sequence structure for summary, documentation, etc., are [5, 6,7,8]. These methods use various approaches to convert the source code to graph structures such as Abstract Syntax Trees (AST), data flow graphs, control flow graphs, paths in AST, augmented AST, contextual flow graphs (utilizing the concepts of Program Dependence Graph (PDG), etc. However, all these approaches do not consider aligning the code with information extracted from texts, images, etc. Hence, the graph structure generated is not scalable for effective super knowledge graph construction.
In code2graph, the knowledge graph extraction from the code poses several challenges such as:

1. The knowledge graph generated from the code needs to encompass necessary information to be converted back to code during graph2code while still being abstract enough to be aligned with the knowledge graph generated by text2graph, image2graph, etc.

2. The knowledge graph needs to be generated for various scientific code which might use different libraries, development environment, etc. For inference task the generated knowledge graph must be abstracted across all the scientific code dataset.

One of the advantages of using the TensorFlow runtime library is that it by default converts all the user programs written in Python that implement the scientific papers into the dataflow graph. Some of the scientific papers in the dataset utilize the TensorFlow runtime which is a cross-platform library where all the computations are represented as a dataflow graph. TensorFlow utilizes the concept of client and master, where client consists of a user level Python program and the master consists of the core runtime kernel implementations (written in C++). The scientific code is written using the client TensorFlow program like Python, converted to the TensorFlow computation graph, and sent to the master as graph definition defined using the protocol buffer. These protocol buffers are used to convert the graph definition into a serialized structure data which are independent of the language and the platform. The client program then creates a Session to execute the computation graphs in a distributed manner on the CPU and GPU available in the system.

These dataflow graph consists of nodes represented by the computation. Moreover, the edges also show control dependencies among various computation nodes. At the fundamental level, every operation is represented as a node and every Tensor is represented as an edge, coming in or out of the nodes, in the dataflow graph. As an example, let’s take the following operation:

```
1. import tensorflow as tf
2. x = tf.constant([[37.0, -23.0], [1.0, 4.0]])
3. w = tf.constant([[11.0, 4.0], [2.0, 0.3]])
4. y = tf.matmul(x, w)
```

*Code Snippet 1: the simple example of TensorFlow graph.*

Here $y = x*w$. To calculate this operation, TensorFlow generates the following dataflow graph for the above operation.

*Figure 1: A simple visualized example of TensorFlow graph using Tensorboard*
However, the codes written in Python that implement in scientific paper consists of various contextual information (such as input data format, extra Python function for pre-preprocessing, etc..) which may not be present in the dataflow graph. Hence, in the first task we create a pipeline to convert the codes to a graph. The graph structure considered in this case is an RDF graph which are used in most of the knowledge graph. This knowledge graph will be used to create the super graph later.

2 Proposed Methodology

After completing the two milestones, the pipeline for extracting the knowledge graph from the code has seen consistent evolution in terms of approaches. In this section, we will highlight the major direction taken and the changes that have been added to extract the knowledge graph from the code.

The pipeline for the knowledge graph extraction is shown in Figure 2. Initially in milestone 1 and 2, we focused on extracting the knowledge graphs through computational graph-based approach. However, we have also explored the possibility of extracting them from static call graphs. We call this the Abstract Syntax Tree (AST) approach, which is highlighted in blue. The AST is a tree which captures the syntactic structure of the source code for the specific programming language. Since the AST represents just the structural content, it is called abstract. An AST data structure is normally used by compilers to represent the structure of the program. In the subsequent section we will explain the methodologies in both pipelines in detail.

2.1 Preprocessing

The datasets such as zziz [9] and paperswithcode [10] contain published papers that have their codes available in open-source platforms such as GitHub. In order to extract the knowledge graph from these source codes, first, we need to check if they use programming frameworks such as Tensorflow and Keras. Second, we need to extract the knowledge graph which is independent of the coding style and library dependency used by the authors to capture the essence of the deep learning architecture. To synchronize with other modules in the DCC project, we mainly used a dataset provided by Siemens, which contains 100 papers with valid links pointing to their GitHub code repositories. There are several steps carried out to perform the pre-processing so that both the computational and the light-weight Abstract Syntax Tree (AST) based method can be utilized to extract the knowledge graph. These steps are explained as follows:
2.1.1 Python Version Check

The code repositories contain codes that are written in Python version 2 and 3. In order to check the version of the Python “py_compile” is utilized. This module is originally used for sharing the Python modules without revealing the source code. Hence, it can be used to pre-check the source files and verify the syntax used in the coded. For each of the code repository, this module is utilized to determine the Python version. Then a virtual environment is created with the specific version of the Python to further extract the information from the code. The code snippet for checking the version is as follows:

```python
1. #!/usr/bin/env python2.7
2. import sys, compileall
3. result = compileall.compile_dir(sys.argv[1], force=True)
4. print("Result:", result)
```

*Code Snippet 2: the script that demonstrates how to programmatically compile Python programs.*

2.1.2 Resolving requirements

After acquiring the virtual environment, the necessary packages required for running the code repository is installed. Normally, these requirements are listed in the README.md file of the GitHub or provided as a “requirements.txt” file. If it is provided as a “.txt” file, then “pip install requirements.txt” is run in the console to re-create the project environment. As mentioned earlier, there are two paths in which the graph is extracted from the code-repository. In the computational graph-based approach, the compiled computational graph needs to be extracted. Hence, in such situation being able to run the Python code until the computational graph compilation is crucial. In the light-weight method we aim to extract the knowledge graph from the static call graph. Although it does not need compilation, issues like version of Python can affect the AST. Hence, resolving the requirements can aid for both the methods.

In situations, where the “requirements.txt” is not provided the parsed AST of the code is analyzed. AST consists of all “import” and “import from” instructions as nodes. These nodes can be parsed to find out the necessary packages and searched in the Python package index (https://pypi.org/pypi/[package name]/json) to get the detailed information of the packages for installation. The code snippet for acquiring the import arguments is shown as follows:

```python
1. tree = ast.parse(code)
2. for node in ast.walk(tree):
3.    if isinstance(node, ast.Import):
4.       for subnode in node.names:
5.          raw_imports.add(subnode.name)
6.    elif isinstance(node, ast.ImportFrom):
7.       raw_imports.add(node.module)
```

*Code Snippet 3: the script that utilizes AST walk to iterate all the import statements in a Python program.*

2.1.3 Code Injection

To extract the computation graph from the codes, the codes need to be partially compiled up-to-the point where the necessary deep learning architecture components have been added to the Tensorflow graph instance. In Keras, the developers usually call a “compile” method before the training starts. However, for the developers who use the native TensorFlow APIs, this is not the case. Instead, the sess.run() should be
found as the computational graph must be compiled before it can be called. In practice, using the ast helper function NodeVisitor, we can search and examine each Call instance in the codes and verify whether its function name is “compile” or “run” (Code Snippet 4). Once found, we then modify the ASTs by programmatically injecting a small portion of code into these locations (Code Snippet 6). Finally, we use the function “to_source” in a Python library astor to re-generate the source code (Code Snippet 5).

```python
1. class CompileVisitor(ast.NodeVisitor):
2.     def __init__(self):
3.         self.found = False
4.     
5.     def visit_Call(self, node):
6.         if isinstance(node.func, ast.Attribute) and node.func.attr == "compile":
7.             self.found = True

Code Snippet 4: the class definition of NodeVisitor that inspects whether an instruction calls "compile".

```python
1. def inject():
2.     inject_code(CompileVisitor, code_tree)
3.     print(astor.to_source(self.code_ast_tree))

Code Snippet 5: the script that demonstrates the steps to perform code injection.

```python
1. def inject_code(class_visitor, node):
2.     location_to_be_inserted = []
3.     
4.     for idx, obj in enumerate(node.body):
5.         visitor = class_visitor()
6.         visitor.visit(obj)
7.         if visitor.found:
8.             if isinstance(obj, (ast.Expr, ast.Call)):
9.                 location_to_be_inserted.append(idx)
10.            elif isinstance(obj, (ast.If, ast.For)):
11.                inject_code(class_visitor, obj)
12.            else:  # should not reach
13.                print(astor.dump_tree(obj))
14.        
15.     if len(location_to_be_inserted):
16.         for idx in location_to_be_inserted[::-1]:
17.             node.body.insert(idx+1, self.inject_code_ast_tree)

Code Snippet 6: the script that demonstrate how to inject the code in the desired positions.

To extract the serialized version of the models in code repositories, we need to inject the script, which is shown in Code Snippet 7, to the codes. In the script, we firstly acquire the current session using Keras. We create a writer module with the current session’s graph and direct it to a location we want to save the file to. The writer module is explained in detail in the next section. At the end of the script, we exit the program. This part is crucial since if we let the program to continue, it will start training and dramatically increase the complexity in pre-processing the codes.
Department of Electrical Engineering
and Computer Science,
University of California, Irvine

1. from keras import backend as K
2. import tensorflow as tf
3. import sys
4. sess = K.get_session()
5. writer = tf.summary.FileWriter([log path], sess.graph)
6. sys.exit()

Code Snippet 7: the script that can be injected to extract TensorFlow graph in TensorFlow programs.

2.1.4 Challenges on Automating the Re-creation of Environment

So far, we have described the functional components to pre-process the dataset. Those components can be integrated to automate the extraction of RDF graphs. However, the code repositories might be written by different developers that have different programming styles and different assumptions, thus making the automation of the whole process harder to achieve.

The main challenges we might encounter during the process, will be to search for the main portion of the code repository, which can exist in different files. In our approach, we search for the files that have the instruction “if __name__ == “__main__”:”, which gives us the hint that this source file contains runnable script. Sometimes, “main.py” exists, and contains the main code that will execute the whole pipeline of the deep learning applications. However, sometimes the DL developers might write multiple scripts that compile their proposed DL models differently. In this case, we need manual effort on skimming through and understanding the whole scientific paper then only can we can determine which model is the one used in the paper.

The second challenge will be the parameterization issue. To extract the TensorFlow graph from the summary writer, the codes need to be invoked. However, some of the hyperparameters used to compile the DL models might be given through the parameters when running the scripts. For example, the developers might design their scripts in this manner “python -input 6 -output 3 -hidden_layers 5 -activation relu first_model.py” so that they can tune their models by simply changing the parameters and record the golden setting somewhere such as readme file. Scanning through the “readme.md” under GitHub repositories might give us some hints on the way to run the script. However, unfortunately, some developers might just not include this information. Therefore, it still requires manual effort to find out how to run the script, which is an easy task for humans.

To automate the full process in recreating the environment we can note the ways that can successfully invoke the scripts so that we can use them to reproduce the result later. Besides, to avoid invoking the codes, we also explore the Lightweight graph extraction approach that can generate RDF graph using static code analysis in parallel. We will describe both approaches in the later sections.

2.2 Computational Graph-based Approach

2.2.1 Dataflow Extraction:

To be able to store the deep learning architectural information from code, we first went through the TensorFlow GitHub repository to find out a built in function named tf.summary.FileWriter [3], which allows us to extract the summary of the computation graph. To extract these summary file, we run the following script:
tf.summary.FileWriter("/tmp/log/...", sess.graph)

The above script takes the directory of the log file and the TensorFlow graph object to be stored. This stored graph is read by the Tensorboard for the visualization purpose as well. In the next step we will extract the graph protocol buffer (tf.graphDef()) which consists of the deep learning architectural information.

### 2.2.2 Protocol Buffer Parser

In order to parse the event file stored by the tf.summary.FileWriter(), we traced the module Tensorboard from the GitHub repository and located the function that understands the event file. The module which will allow us to parse event files is called event_accumulator which can be imported from Tensorboard as follows:

```python
from tensorboard.backend.event_processing import plugin_event_accumulator as event_accumulator
```

Then, we found out that we can extract the protocol buffer using the following script:

```python
accumulator = event_accumulator.EventAccumulator(logdir)
 accumulator.Reload()
 graphDef = accumulator.Graph()
```

The structure stored in graphDef is a serialized version of the protocol buffer. We need to parse graphDef object to process it and extract the architectural structure. The best option to see the hierarchical structure of the model architecture is to convert the graphDef object to JSON format. In order to convert the graphDef to Json we have utilized a built-in module called MessageToJson. Protocol buffer has a method built in called MessageToJson under json_format. (google.protobuf.json_format.MessageToJson(...)) The details of json_format module can be found from Google Protobuf Documentation website at [4]. A glimpse of the function prototype is below:

```python
def MessageToJson(message, including_default_value_fields=False):
    '''
    Converts protobuf message to JSON format.
    Args:
    message: The protocol buffers message instance to serialize.
    including_default_value_fields: If True, singular primitive fields, repeated fields, and map fields will always be serialized. If False, only serialize non-empty fields. Singular message fields and one of fields are not affected by this option.
    Returns:
    A string containing the JSON formatted protocol buffer message.
    '''
```

With this function, we have acquired the JSON string and converted it to the RDF format. The function to do this in our code repository is json2rdf, which is explained in later portion of the report. In practice,
MessageToJson returns the string of stored JSON message. Thus, the following code example, which incorporates Python json library for encoding and decoding json format data, was adopted:

```python
def convertGraphDef2Json(self, graphDef=None):
    if graphDef is None:
        self._jsonGraph = json.loads(json_format.MessageToJson(self._graphDef))
    else:
        self._jsonGraph = json.loads(json_format.MessageToJson(graphDef))
    return
```

### 2.2.3 Converting Json to RDF Graph:
As for now the JSON string, which has an immensely complex structure, has been extracted by calling the tf.GraphDef(), we then parse its hierarchical structure first in order to convert it into RDF. This JSON string carries a set of nodes, which are essentially the fundamental building blocks of the computational graphs, and a set of attribute-value pairs to represent each node’s properties. For example, the node, with name “dense/bias/Adam/Assign”, has five fields, ‘attr’, ‘input’, ‘device’, ‘name’, and ‘op’. Among the fields, the ‘attr’ carries the necessary information for performing the computation, while the ‘input’ has the list of nodes that output tensor values to this node. The ‘op’ is the name of the type defined in TensorFlow C++ run time library. To be able to parse the graph, a parser must be created to iterate through the nodes as well as their attribute-value pairs. We have implemented a class structure that is responsible for this called Json2RDFParser.

```json
{  
    'attr': {  
        '_class': {'list': {'s': ['...==']}},  
        'dtype': {'type': 'DT_FLOAT'}},  
    'input': ['dense/bias/Adam', 'dense/bias/Adam/Initializer/zeros'],  
    'name': 'dense/bias/Adam/Assign',  
    'op': 'AssignVariableOp'
}
```

The procedures to parse the JSON graph consists of the following: parsing node hierarchy, parsing node device, parsing node attr and parsing node input, each of which is done in a separate procedure. We have utilized a Python library for rdf representation and processing called rdflib [6]. Once the parsing task is done up to this point, another procedure called simplifying the graphs was called to process the resultant graphs to a simpler version. This simpler representation, which aims to contain only deep learning architectural information, then can be further utilized to polish the universal RDF representation created by other modules in DCC.

### 2.2.3.1 Parse Node Hierarchy
To create the RDF graph, we first constructed the hierarchy of the nodes. In this hierarchy there are nodes which are described in the computation graph, and the ones that do not have definition but are the parent node of multiple operations. We call these type of nodes, virtual nodes. Using these nodes, allowed us to represent the architecture of deep learning algorithms. Although, it might be intuitive to assume that all the nodes should be connected to the Model (or Root), the complexity of the RDF graph will become intractable as the complexity of the deep learning algorithm increases. In addition, the extracted hierarchical information can be represented with different resolution (in terms of depth from the root node). Therefore, the first objective is to get the sub-components of this hierarchical structure and collapse these components as needed to determine the resolution of the extracted RDF graph.
Coming back to our example, “dense/bias/Adam/Assign”, while iterating to find “/”, we determine if this node can actually be expanded to four separate nodes carrying hierarchical information, such as “dense”, “dense/bias”, “dense/bias/Adam” and “dense/bias/Adam/Assign. The corresponding RDF-formatted triples are added as follows.

1. Format: <name of parent, RDFS.member, name of children>
2. Example 1: <Root, RDFS.member, dense>
3. Example 2: <dense, RDFS.member, dense/bias>
4. Example 3: <dense/bias, RDFS.member, dense/bias/Adam>
5. Example 4: <dense/bias/Adam, RDFS.member, dense/bias/Adam/Assign>

2.2.3.2 Parse Node Component: “device”/”input”
After getting the node structures, we start to add the triples to RDF graph. For the nodes that carry out the operations, we iteratively check each node, which is basically a dictionary architecture, and look for the key ‘device’ and “input”. If it has it, we add its value with the node name and hierarchy as a triple to our RDF list. If the key ‘device’ cannot be found, it means that the node in question does not specify which device to run on (GPU or CPU). In addition, if the key ‘input’ cannot be found, the node itself will perform the specific operation without processing any input. The corresponding RDF-formatted triples are added as follows.

1. Format: <<node_name>, has_input/on_device, <node_name>>
2. Example 1: <dense/bias/Adam/Assign, has_input, dense/bias/Adam>
3. Example 2: <dense/bias/Adam/Assign, has_input, dense/bias/Adam/Initializer/zeros>

2.2.3.3 Parse Node Component: “op”
Like the previous step, we get the operation names for each component and add them to our RDF structure. Operation names are saved under the keyword ‘op’ in JSON string, so we look for that and add them when found using predicate “has_op”. For later usage, the has_op can be used in query to identify whether an entity in a RDF graph is a high-level API node or a node that carries out the unit operation. The corresponding RDF-formatted triples are added as follows.

1. Format: <<node_name>, has_op, <op_name>>
2. Example 1: <flatten_input, has_op, Operation_Placeholder>
3. Example 2: <dense/bias/Adam/Assign, has_op, Operation_AssignVariableOp>

2.2.3.4 Parse Node Component: “attr”
Like the inputs, each component also has attributes that is required for data flow graph to be run. The problem with this step is that attributes themselves can have attributes. This goes on and on. Thus, to parse attributes of each node and add them to our RDF list, we had to do a recursive search for attributes. Also, while doing this recursive search we had to keep the hierarchical information of the attributes as well. It was crucial for us to understand what attributes are necessary for each of the operations while converting the RDF graph back to the computational graph format. In order to do this, the parser code first recursively populates the type of attribute for each of the operation while going through the dataset, generating a list of attributes while traversing from node's 'attr'. An example of generated attribute list is shown below:

1. { 'attr': { '_class': { 'list': { 's': ['$=='] } } } },
A glimpse of the realized pipeline is shown below,

```python
for node in nodes:
    if 'attr' in node.keys():
        attr_paths = self.recursive_scan_attr(node['attr'], path=[])
        sb = BNode(node['name'])
        for attr_path in attr_paths:
            if not sum([1 if rule.parse(attr_path) else 0 for rule in self.rules]):
                print("can not recognize %s % str(attr_path))
                raise Exception("Unknown attribute")
    for rule in self.rules:
        results = rule.get_results(sb)
        if results:
            self.RDF_graph += results
```

In order to parse the attributes, we have implemented the rules using class data structure based on the structure of the node representation in the computational graph, whereby each of the rules has its own class. With a list of attributes as the input, we utilize the rules to filter the generated list and construct the RDF triple. If the defined rules cannot parse the generated list, we receive an exception from the code. This exception is handled manually to generate more rules to parse the list. This was easier for us than having to go through the whole tensorflow GitHub repository to just extract the meaningful attributes. In addition, with the rule class, we have further defined how to turn RDF triple back to a key pair under 'attr' for later graph2code usage. An example of final RDF triple is as follows:

```
Format: <<node_name>, <attribute_name>, Literal<value>>
Example 1: <dense/bias/Adam/Assign, dtype, "DT_FLOAT">
Example 2: <dense/bias/Adam/Assign, _class, "...=="/>
```

A glimpse of the type rule generator is as below:

```python
class DTypeGenerator(GGenerator):
    def __init__(self):
        self.attr_uri = URIRef("http://example.org/dtype")
        self.value = None
        self.parsed = False
    def parse(self, attr_list):
        if attr_list[0] == 'dtype':
            self.value = attr_list[1]
            self.parsed = True
        return True
```
In general, by going through the pipeline mentioned until now, a completed RDF graph for representing a TensorFlow/Keras based deep learning architecture can be generated. However, for the purpose of getting the deep learning architecture information, this graph is still far too complicated. Thus, the simplifying mechanisms over the RDF graph will be described in the following section.

2.2.3.5 Simplify the resultant RDF graph
As mentioned earlier, with all the attributes RDF graph is too complicated to be used in applications. There are usually thousands of triples in each RDF (which varies based on different deep learning models). In this step we demonstrate how we have solved this problem by compressing our RDF graph. The compression is done by a rule-based breadth first search starting from the Root. We searched and expanded the nodes that have RDFS.member as a relation. A glimpse of the code snippet that carries out this task is shown below. We have created a dictionary with keywords for “unnecessary” and “interested” node names etc. based on the ontologies extracted from the TensorFlow GitHub repository. In this context, unnecessary means that they do not give any critical information about the deep learning architecture used in the paper. If the attributes are in the unnecessary category, we combine them with the ones in the higher level of the hierarchy.

After the BFS, words_bank contains the nodes that will be presented in the simplified RDF graph. We then start to construct the simplified RDF graph by adding the nodes that are in the words_bank. One thing worth mentioning is that, while compressing the nodes, the information in the original RDF graph should still be maintained. For example, in order to maintain the consistent relationship of inputs and outputs between two
high-level nodes, we had to check if there is any pair of its lower level nodes that has an input-output relation. The above-mentioned keyword-based method can assist us to acquire simplified RDF graphs by giving a list of general machine-learning terms and unwanted procedural terms.

2.3 Ontology and Vocabulary Extraction

Before delving into the approaches that produce the RDF knowledge graphs, the ontology is needed. TensorFlow has a well-defined structural definition which can aid in describing the DL architectures (such as a layer, loss, optimizer, input, output, etc.). Therefore, to extract the hierarchical structure of the DL architecture, we first extracted the ontology from the TensorFlow Official Website. Although tf.GraphDef() naturally have these definitions, it consists of extra computation graph information. The ontology helps us better understand how to convert DL codes into RDF knowledge graphs and how to separate the architecture from the computation graph metadata necessary for the TensorFlow core C/C++ runtime. Hence, to aid us in focusing on DL architecture we have extracted the TensorFlow API module hierarchy as the ontology (see Figure 3). It can be noticed in Figure 3 what types of nodes to focus on extracting can be listed by populating the custom vocabulary with leaf nodes (which are the callable instances used in the Python code while creating the DL architecture). For example, “tf/keras/estimator/model_to_estimator” is a leaf node while “tf/keras” is not. All the non-leaf nodes are treated as virtual nodes in the RDF graph for us to better present the structure of the DL architecture present in the code. Some more examples on the hierarchical information of TensorFlow used in ontology generation is shown in Figure above. It can be also be visualized using Python plotting libraries such as networkx as below.

![Ontology structure extracted from the TensorFlow repository. It can be noticed that all the instances are arranged in tree structure (for example Dense, Conv1D, Dropout are children of Layer node, where “layer” is a virtual node).](image-url)
In practice, a class TFVocScraper defines the logics to scrape and parse the TensorFlow official Website. Besides, we implemented OntologyManager to associate a string keyword to its semantic meaning on the TensorFlow ontology. To realize the functionalities, the “fuzzy search” function will compare and match the name scopes between queried string and terms stored in ontology_manager and generate a ranked list of the most related terms in the ontology. With the ranked list, we then select the most relevant one and to construct the relation by creating the triple <s, p, o> with the RDF.type as its predicate. Few examples are shown below,

```
1. >> ontology_manager.fuzzy_search("Dense")
2. ('tf.layers.Dense', {'name': 'tf.layers.Dense', 'url': '/versions/r1.13/api_docs/python/tf/layers/Dense'})
3. >> ontology_manager.fuzzy_search("dense")
4. ('tf.layers.dense', {'name': 'tf.layers.dense', 'url': '/versions/r1.13/api_docs/python/tf/layers/dense'})
5. >> ontology_manager.fuzzy_search("relu")
6. ('tf.nn.relu', {'name': 'tf.nn.relu', 'url': '/versions/r1.13/api_docs/python/tf/nn/relu'})
```

2.4 Light-Weight Method

In this section, we will describe the approach to generate RDF graphs without invoking the scripts. In milestone 1 and 2, the primary focus on extracting the graph was dependent on being able to acquire the compiled computational graph. However, this approach requires partial compilation which may be ineffective for scaling the graph extraction process across different code repositories. Hence, in the light-weight method we explore methods in which compilation of the code repository will not be necessary. While writing the script that realizes their deep learning architecture, most of the time the developers invokes Tensorflow APIs or Keras APIs in order to make the codes simple and to make the debugging process easier. Thus, tracking the occurrence and its order of such TensorFlow API invocation can assist us to construct the computation graph of the models. The approach is essentially a deep code finder that syntactically traverses the codes by looking for keywords. The steps consist of generating the call graph, generating the call tree, lightweight RDF graph generation then generating the TensorFlow word sequence.

2.4.1 AST Extraction

As mentioned earlier, an Abstract Syntax Tree is a data structure which captures the syntactic structure of the source code for the specific programming language. An AST data structure is normally used by compilers to represent the structure of the program. AST trees are extracted using an ast [11] module. The AST data structure consists of the important information such as implemented modules and methods in the deep learning architecture. Utilizing the ontologies and vocabulary, we can extract the deep learning architecture information (such as layers types, activation function, optimizers, etc.) implemented in the architecture.

2.4.2 Call Graph Extraction

AST data structure only consists of the syntax grammatical structure. It does not necessary consists of the architectural information. Hence, to get the program structure and architecture information, a static call graph is generated. This is done utilizing the pyan [12] library. After the call graph is generated for a DL project, we determine the starting points by finding the zero in-degree vertexes on the call graph, which means no other modules or functions will invoke this function in the project. After finding the starting
points, we then start to inspect the body of the function by calling the AST helper function “iter_child_nodes”. The NodeVisitor cannot be used here because it will show the order of the function calls in the source files instead of in the execution. Whenever the child node is simply an expression, we then use TFCallVisitor to identify whether the function calls are the TensorFlow function calls or the normal function calls. The way we determine whether the function calls are TensorFlow function calls is to use the search function defined in the ontology manager. If the found function calls are normal functions, we will check the AST trees stored in the memory and then inspect it recursively. When visiting the function calls, the parameters and the keywords are also the targets to be searched because sometimes some important DL architectural information may be assigned in keywords or parameters. By inspecting the program in this line-by-line manner, we then can build the call trees for the projects. Usually, some generated call trees start from the functions written for testing, this might not be invoked in the main routine. In this case, we can filter those call trees which have no TensorFlow function calls.

![Figure 4 Sample call graph](image)
The call trees contain both architectural and sequential information for the codes. In the following sections, we then can use call trees to generate both RDF knowledge graphs and TensorFlow keyword sequences.

### 2.4.3 Graph Parsing

With the call trees extracted, we then generate the RDF graphs to represent the DL model. For each node in the call tree, for TensorFlow function type, we create a triple with the object to be its corresponding URL while for the normal function calls, we create triples that contain their types (Module or Function or Method) in the source files. The name of the node includes not only the information of the calling path starting from the root but also the sequential order of the function calls. For example, the node with the name “.testGraph_extensive.Sequential_2” means that the Sequential is the second function call when running the main routine in the module testGraph_extensive. For arguments and keywords, we add the triples with the corresponding keyword and the order of argument encoded in the predicate. Whenever the node contains children nodes, we then recursively call the generation routine.

### 2.4.4 Sequence Extraction

To generate the TensorFlow sequence for the codes, we can use the same traversing mechanism when generating the lightweight RDF graphs. Whenever the node shows that it is a TensorFlow function call, we then add this expression to the list of TensorFlow keyword sequence. Whenever there exist keywords and arguments, we then append those to the list as well. If the node has children nodes, we then recursively call the generation routine. In this manner, we can generate the keyword sequence with each keyword in the sequence is ordered according to the call trees instead of the literal occurrences on the source files.

### 3 Code Implementation Detail

#### 3.1 Computational graph-based Approach

For all the pipeline mentioned above, we have implemented a general class called graphHandler as shown below:

```python
1. class graphHandler(graphMETA):
2.     def __init__(self,
3.                 graphDef = None,
4.                 RDF = None,
5.                 jsonGraph = None,
6.                 graph = None,
7.                 tradGraph = None,
8.                 logdir = None):
9.         '''Initializing the class'''
10.        self._graphDef = graphDef
11.        self._RDF = RDF
12.        self._jsonGraph = jsonGraph
13.        self._graph = graph
14.        self._tradGraph = tradGraph
15.        self._logdir = logdir
16.        pass
```

Under `graphHandler` class, `graphDef`, `RDF`, and JSON formats are saved as properties. With the `EventAccumulator` function from Tensorboard, mentioned earlier, we have extracted the `tf.graphDef()`
object of a given machine learning model. Input of this function is the directory of event file resides. To use
this function, we have implemented a method for our class called readGraphDef, which takes the directory
path as input and saves the extracted graphDef as a property _graphDef. The method definition is given
below:

```python
1. def readGraphDef(self, logdir=None):
2.     '''function to read graph from drive'''
3.     if logdir!=None:
4.         pass
5.     else:
6.         accumulator = event_accumulator.EventAccumulator(self._logdir)
7.         accumulator.Reload()
8.         self._graphDef = accumulator.Graph()
9.     return
```

The _graphDef property will now consist of the serialized version of the proto-buffer that was
extracted from the event file. It can be visualized by printing it out using our method displayGraphDef.

```python
1. def displayGraphDef(self, graphDef=None):
2.     '''function to display the graph'''
3.     if graphDef!=None:
4.         #TODO: write a function to display graph
5.     else:
6.         #print(self._graph)
7.         for node in self._graphDef.node:
8.             print(node)
9.     return
```

Previously, we have mentioned that graphDef needs to be converted to JSON type to be analyzed. Our
method convertGraphDef2Json takes care of that. It saves the JSON string under its property _jsonGraph.
This JSON string can be saved as a JSON file using json.load() function from JSON library. The method
definition is given below:

```python
1. def convertGraphDef2Json(self, graphDef=None):
2.     if graphDef==None:
3.         self._jsonGraph = json.loads(json_format.MessageToJson(self._graphDef))
4.     else:
5.         self._jsonGraph = json.loads(json_format.MessageToJson(graphDef))
6.     return
```

To convert the JSON file to RDF structure, we call the method convertJson2RDF. It then, calls json2RDF
function. json2RDF function uses a class we defined called Json2RDFParser, as mentioned earlier, to parse
the JSON string and convert it to RDF format. The method convertJson2RDF is given below:

```python
1. def convertJson2RDF(self, jsonGraph=None):
2.     '''function to convert json to RDF format'''
3.     if jsonGraph!=None:
4.         self._jsonGraph=jsonGraph
5.
```
The Json2RDFParser class takes the extracted JSON string and parses it. It creates triples according to the RDF structure. All the steps from the previous chapter have been implemented as methods to this class. In the end json2RDF function creates a Json2RDFParser object and uses its parse method to call all the methods responsible for parsing the JSON string.

### 3.2 Lightweight Approach

For all the pipeline mentioned section 2.4, we have implemented a class called TFTTokenExplorer to include the functionalities as shown below,

```python
class TFTTokenExplorer:
    def __init__(self, code_repo_path):
        self.code_repo_path = code_repo_path

    # mapping utilities
    def call_graph_visitor = CallGraphVisitor(glob("%s/**/*.py" % str(code_repo_path), recursive=True))
    self.pyan_node_dict = {} # hashmap from RDF node name to pyan node.

    # generated in build the one complete call graph
    self.call_graph = Graph()

    # generated in build call trees
    self.call_trees = {}

    # generated in build rdf graphs
    self.rdf_graphs = {}

    # generated in build tfsequences
    self.tfsequences = {}

def explore_code_repository(self):
    self.build_call_graph()
    self.build_call_trees()
    self.build_rdf_graphs()
    self.build_tfsequences()
    self.dump_information()
```

Under the TFTTokenExplorer, the attributes call_graph_visitor and pyan_node_dict are dictionary-like instances that help look up the corresponding node or instance when inspecting a function call, while the others are used to cache and store the intermediate results for debugging purpose. The main routine that realizes the lightweight based graph extraction approach is “explore_code_repository” and consists of four major steps: building the call graph, building the call trees, building the RDF graphs and then generate the TensorFlow keyword sequences. The function “dump_information” not only prints an abundance of logging information that helps to debug when generating the RDF knowledge graphs but also visualizes the lightweight RDF knowledge graphs using a Python library pyvis.
The structure of the function “build_call_trees” will be as follows: we first use the call graph generated by pyan to find zero in-degree vertex, and then start to generate the corresponding call tree for each of them. In the process, the function “grow_function_calls” incorporates the ProgramLineVisitor, which performs most of the static code-inspecting behaviors to generate the call trees. The call tree itself is a multivariate tree structure where each node is a Python dictionary and owns a list of children.

```python
1. def build_call_trees(self):
2.     roots = self.find_roots(self.call_graph)
3.     for root in roots:
4.         call_tree = self.grow_function_calls(root)
5.         call_tree["rdf_name"] = call_tree["name"]
6.         self.populate_call_tree(call_tree)
7.     self.call_trees[root] = call_tree
```

For visualization purpose, we wrote a function “populate_call_tree” to populate the name scopes from parent nodes to children nodes. For fashion MNIST example, the node “Flatten” will have a “rdf_name” which is “.test_extensive.Sequential_0.Flatten_2” after calling the function “populate call_tree”. The structure of functions “build_rdf_graph” and “build_tfsequence” are similar in the sense that both of them will recursively traverse the call tree and visit the attributes for each node in the same order (“type”, “args”, “keywords”, “children”).

4 Results

4.1 Computational Graph Extraction

4.1.1 Test on simple graph
The raw computational graph extracted from the MNIST example is shown as follows:

![Figure 5: Raw computational graph extracted from MNIST example.](image-url)
As mentioned earlier, with all the attributes RDF graph is too complicated to be used in applications (see Figure 5). There are usually thousands of triples in each RDF (which varies based on different deep learning models).

The mentioned keyword-based method used to simplify the RDF graph, however, has a restriction which is that a user might define the name spaces by themselves, so the proposed compression method will work only under the computational graph where users do not have their own defined namespaces. Tensorflow type aware method is one possibility that BFS process will stop at the level where the node can be recognized from Tensorflow/Keras API. For example, a bunch of computation that carry out for the first layer “dense” on the above screenshot can be compressed by binding its type to tf.keras.layers.Dense. In this manner, it will not only make the simplified RDF graph more intuitive to represent the deep learning architecture, but also assist the process of transforming the graph back to code template which is a later milestone to achieve in phase 2.

4.1.2 Test on real scientific paper’s code repository

Figure 6: Simplified RDF graph for the MNIST fashion classifier.

Figure 7: Simplified RDFs extracted from the code repository. Left RDF is for paper number 78 (7-layer CNN model with tanh activations, located at [darpa_aske_dcc] / src / paperswithcode / data / 78 in the gforge repository) and right RDF is for
paper number 24 (Few shot 3D-Unet architecture based on Generative Adversarial Network, located at [darpa_aske_dcc] / src / paperswithcode / data / 24 in gforge repository).

<table>
<thead>
<tr>
<th>Paper ID</th>
<th># of nodes</th>
<th># of has_input</th>
<th># of has_op</th>
<th># of on_devcie</th>
<th># of attr</th>
<th># of node</th>
<th># of has_inputs</th>
<th># of has_ops</th>
<th># of on_devcies</th>
<th># of attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>998</td>
<td>882</td>
<td>687</td>
<td>3</td>
<td>2394</td>
<td>127</td>
<td>83</td>
<td>66</td>
<td>0</td>
<td>197</td>
</tr>
<tr>
<td>30</td>
<td>932</td>
<td>984</td>
<td>705</td>
<td>699</td>
<td>2283</td>
<td>287</td>
<td>191</td>
<td>219</td>
<td>219</td>
<td>750</td>
</tr>
<tr>
<td>31</td>
<td>1475</td>
<td>1553</td>
<td>1003</td>
<td>0</td>
<td>3043</td>
<td>36</td>
<td>28</td>
<td>16</td>
<td>0</td>
<td>97</td>
</tr>
<tr>
<td>37</td>
<td>2122</td>
<td>1557</td>
<td>1641</td>
<td>0</td>
<td>6216</td>
<td>756</td>
<td>603</td>
<td>618</td>
<td>0</td>
<td>2343</td>
</tr>
<tr>
<td>42</td>
<td>17246</td>
<td>21311</td>
<td>11026</td>
<td>3</td>
<td>37970</td>
<td>1446</td>
<td>1786</td>
<td>1196</td>
<td>0</td>
<td>4488</td>
</tr>
<tr>
<td>43</td>
<td>6372</td>
<td>6966</td>
<td>4208</td>
<td>3</td>
<td>12187</td>
<td>315</td>
<td>321</td>
<td>240</td>
<td>0</td>
<td>913</td>
</tr>
<tr>
<td>49</td>
<td>700</td>
<td>676</td>
<td>397</td>
<td>3</td>
<td>1276</td>
<td>116</td>
<td>74</td>
<td>64</td>
<td>0</td>
<td>260</td>
</tr>
<tr>
<td>52</td>
<td>5922</td>
<td>6316</td>
<td>4080</td>
<td>0</td>
<td>12713</td>
<td>456</td>
<td>463</td>
<td>356</td>
<td>0</td>
<td>1324</td>
</tr>
<tr>
<td>58</td>
<td>25495</td>
<td>383839</td>
<td>15538</td>
<td>3</td>
<td>42870</td>
<td>443</td>
<td>920</td>
<td>240</td>
<td>0</td>
<td>710</td>
</tr>
<tr>
<td>78</td>
<td>535</td>
<td>375</td>
<td>319</td>
<td>0</td>
<td>904</td>
<td>44</td>
<td>20</td>
<td>14</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>94</td>
<td>345</td>
<td>237</td>
<td>211</td>
<td>0</td>
<td>701</td>
<td>42</td>
<td>21</td>
<td>13</td>
<td>0</td>
<td>41</td>
</tr>
</tbody>
</table>

| General/deep-rdp | 830 | 858 | 499 | 3 | 1550 | 214 | 149 | 143 | 0 | 528 |
| General/danq     | 919 | 912 | 719 | 0 | 2410 | 503 | 491 | 419 | 0 | 1381 |
| General/FIDDLE   | 1921| 160 | 1320| 0 | 4395 | 320 | 265 | 219 | 0 | 816  |

Table 1: Statistics of RDF graph extracted using code2graph.

After running the code2graph, the statistics for various attributes before and after simplifying the graph is presented in Table 1. It can be notices that after simplifying the graph, the computation graph’s metadata not necessary for defining the deep learning architecture has been significantly removed. However, the rule-based filtering approach is not yet perfect and needs further tuning.

4.2 Lightweight Graph Extraction

4.2.1 Test on simple graph

The following graph is the visualization of the resultant RDF graph for Fashion MNIST example. In the figure, the green vertexes represent the modules, blue vertexes represent the normal function calls, red vertexes represent the TensorFlow function calls. The light blue vertexes represent the argument or keyword values used during the TensorFlow function calls.

Figure 8: The visualization of the Lightweight RDF knowledge graph.
4.2.2 Test on complicated graph

The test result for a complicated graph is shown as follows:

Figure 9: Light-weight AST-based graph extraction for complicated graph.

Figure 10: VGG16 RDF extraction
Figure 11: RDF graph extraction from Alexnet

Figure 12: RDF extraction from Xception
4.3 Jupyter Notebook Example

4.3.1 Computation graph-based graph extraction
We have prepared a test script to show our code. Our codes have been written and tested with Python3.5 and Python3.6. The step by step explanation of running the test script is given below:

4.3.1.1 Install the requirements
Since we have used various libraries in order for our code to work, they must be installed. All the required libraries can be installed with Python module pip. We have prepared the requirements.txt file and included it in our repository. To install all of them run the following script in your terminal, working directory should be the parent directory:

1. python3 -m pip install -r requirements.txt

4.3.1.2 Run the Jupyter Notebook
By running the following line, the Jupyter Notebook session is started:

2. jupyter notebook

This will open the browser with the notebook environment. From there navigate to testScript folder and open code2graph.ipynb file. This is the notebook we have prepared for test purposes. **Make sure that the kernel is running on Python 3.** It is possible to run our code cell by cell. Run each cell from top to bottom by clicking on “Run”. The cell in the far bottom will display the RDF graph we have extracted from the given event file. It will both create a visualization in a new browser page and print out each triple that has been added to the graph. The notebook page is given below after running the cells.

![Figure 13: Test script walk through in Jupyter.](image)
After running the last cell, an html page also pops up with an interactive RDF graph (drawn using pyvis opensource library [8]) viewer where all the components, the relation among the components can be viewed (see Figure 10).

**Figure 14:** Plotting of simplified RDF graph.

### 4.3.2 Lightweight-based method

The Jupyter notebook for the light-weight AST-base method is located in https://github.com/deepcurator/DCC/tree/development/src/code2graph/testScript

The result of running the notebook for the MNIST example is shown as follows:

```python
In [1]:
    """First, we import the graph creator module and the necessary library for path analysis""'
    # Our program uses some libraries that need to be installed before using it
    # The Libraries are listed in the requirements.txt file.
    # Before running this script go to the main folder of codegraph module.
    # When the working directory is code2graph main folder, run the following line in the terminal:
    # pip install -r requirements.txt
    # After that Python's pip module will install all required libraries.
    # Having all the libraries ready, now we can import our main module for usage

    import sys
    sys.path.append('core')
    from graphlightweight_calltree import *
    from pathlib import Path

    print("Necessary modules have been successfully imported!")

Necessary modules have been successfully imported!
```
In [2]: ```
# Get the path for our fashion_mnist example

# Our module requires a repository path to explore.
# It will find every code file and include it in the analysis.

# For demonstration purposes we have prepared 2 different examples:
# The fashion_mnist classifier example, and a simple function call operation.

# In order to use it on other repositories change this path.
# A path = Path('/', )/core/'text'/'simple_example'
# path = Path('/', )/core/'text'/'fashion_mnist'
path = path.resolve()

print('Resolved Path is:', path)
Resolved Path is: D:\Projects\UCCI\src\code\core\test\fashion_mnist
```
5 Future Work

5.1 Filtering Architecture RDF from unnecessary data
The extracted knowledge graph from computational graph consists of large amount of metadata that is specific to the C++ core kernel, whereas, the light-weight AST based approach has un-necessary metadata about the Python programming language. Due to this, we have focused heavily on filtering extracted graph so that only the knowledge graph relating to the scientific paper’s deep learning implementation is captured. In the next phase, we will utilize these graphs to train our unsupervised learning algorithms to infer various parameters of the RDF graph. However, to improve the inferences, we will continue working on filtering the knowledge graph.

5.2 Generating Code from RDF graph

After the super-graph is created, the RDF graph for the code will be inferred. There are various works that have previously focused on generating the code using various approaches. For example, [15] utilizes the input-output examples of the source code using neural networks, work in [16] utilizes probabilistic grammar model to aid code generation. The closest work that is generating code by converting it to a graph and back is [17]. The authors utilize generative models for encoding and decoding the AST graph. However, our
approach relies on inferred RDF graphs, which require us to be able to generate computational graphs from graphs inferred from alignment of information from multiple sources. The challenges in generating the code from the such inferred RDF graphs are as follows:

1. Analyzing the probabilistic RDF and convert it to an RDF that maximizes the probability of having the same architecture described in the paper.

2. After getting the maximum probability RDF, to be able to execute it, it needs to be converted to the corresponding computational graph.

To be able to execute this RDF graph it needs to be converted to the corresponding computational graph. From this computational graph we intend to extract the client TensorFlow code corresponding to the scientific papers that lack codes. Each of the steps for converting the corresponding inferred RDF graph to the code is explained as follows:

5.2.1 Rule-based preprocessing of the RDF graph:
The inferred RDF graph is probabilistic, which means it cannot be directly converted to the computational graph as some of the suggested nodes in the graph may not be possible to implement in TensorFlow using the dataflow graph. Moreover, the suggested entities of the nodes may be erroneous and may not exist at all. Hence, the first step in converting the inferred RDF graph to code will be to utilize the TensorFlow higher level and lower level API guide to create a dictionary of possible attributes and entities for the individual nodes of the RDF graph. Based on this dictionary the inferred RDF graph will be filtered to add or remove the information to make sure that the inferred RDF can be converted to the code.

5.2.2 Adding context/attribute to the RDF graph:
Once the RDF graph has been filtered, we must add the context and attributes necessary for executing the computational graph corresponding to the RDF graph. For example, as shown in Figure 6, the 2D convolutional API has various arguments whose attributes must be defined. The size of the filters, output shape, data type of the output, etc., are some of the attributes corresponding to the conv2d operation. Moreover, some of the nodes will require specifying types of sub-operations (for example type of initializers used: constant, uniform, random normal, truncated normal, etc..). In order to acquire these attributes and context and add it to the RDF graph, we will first utilize default values for each of the TensorFlow operations. Later, we when the super-graph is aligned and created, we will query the super graph to acquire additional possible attributes if available. Moreover, some of these attributes may just end up being the hyper-parameters used in the deep learning architecture implemented in the scientific paper. We assume that all of the hyper-parameters and attributes will not be possible to extract from the super-graph created from the text, images, equations, etc., without having the code. Hence, some of the attributes will need to be predicted. Therefore, we will perform attribute clustering for corresponding computational operations based on the training dataset to predict the most likely attributes for the given TensorFlow operation (such as type of initializers, data-type, etc.). Moreover, we will utilize generative models to embed the RDF and the computational graph to aid in predicting possible attributes.
5.2.3 **RDF graph to tf.Graph():**

Once the RDF graph has been pre-processed and edited by filling out the missing information, we will first convert it to JSON format and then to the tf.GraphDef() protocol buffer. Since the format of the JSON and the tf.GraphDef() are similar, this conversion will be trivial. Once we have the tf.GraphDef() we can easily acquire its corresponding tf.Graph() by parsing the serialized protocol buffer. This is a crucial step as, once we have acquired the tf.Graph(), we can check if we can create a TensorFlow session from the client side and submit the graph to the master TensorFlow core runtime. However, before submitting the graph to test if it can be run, the data necessary for running the TensorFlow Session needs to be determined. This will be performed by querying the super-knowledge graph.

5.2.4 **Evaluation of Graph2Code:**

The evaluation of graph2code’s performance can be performed at various levels. At the graph level, we can check the structural similarity of the computational graph generated from the inferred RDF graph to the testing computational graph generated from the scientific code test data-set. Moreover, a quantitative analysis can be performed to test if the accuracy/loss metrics proposed and presented as results in the scientific papers match the ones that have been calculated using the computational graph generated from the inferred RDF graph.

5.2.5 **Converting tf.Graph() structure to code:**

Once the computational graph (tf.Graph()) is runnable, we will have recovered the scientific code necessary for implementing the deep learning architectures proposed in the scientific paper. This computational graph can be saved and edited to expand the existing scientific papers to present new and novel architectures (see an example below).

```python
import tensorflow as tf

with tf.Graph().as_default():
    Model.add(keras.layers.Dense(10, activation=tf.nn.softmax))
```

However, this dataflow graph is a language independent representation of the Python code which is used to aid in storing, transferring, and finally re-storing in a C++ program (This is enabled when the tf.Graph() extended with information regarding its associated variables, assets, and set of inputs and outputs from a graph and saved a tf.MetaGraph()). In order to ease the task of editing the deep learning architecture, we will also generate the client side TensorFlow Python code template. In order to do this, we create a parser to go through the tf.GraphDef() and use the high level APIs such as Keras, Estimators, etc., supported by TensorFlow to create the deep learning models, add subsequent components present in the tf.GraphDef() with their corresponding attributes, etc. The code will be generated using the latest library version.
5.3 Knowledge Graph Embedding

In phase 2, the focus of the project will be to acquire a large amount of knowledge RDF graphs from the code repository to train an auto-encoder so we can embed the high-dimensional RDF knowledge graph to a smaller dimension embedding value. We will further explore the auto-encoder architecture to assist tasks such as completing the head or tail entities when either is missing, predicting the relation given the two entities in the RDF triple or creating a sequential model to capture the RDF structure beyond first-order logic. This embedding will then aid in improving the code knowledge graph inferred by the Siemens’ team.

5.4 Similarity Measure and Novelty Measure in Embedding Space

After embedding the knowledge graph triples and the larger part of knowledge graph, in the second phase, we will perform the similarity and difference measurement utilizing the embedding values. The architecture will consist of utilizing the result of an unsupervised model such as the knowledge graph auto-encoder along with other supervised algorithms to acquire better measure of similarity and difference among the knowledge graph of scientific papers.
6 Discussion
The major challenge for extracting the graph from the code was the large amount of variation in the code repositories in terms of Python version, dependent libraries, author’s coding style, to name a few. While the AST based approach, which acquires the static call graph to extract graph structure, is scalable and convenient, it requires dedicated algorithms to extract the sequence structure which sometimes are only available after the code has been fully compiled. On the other hand, the computational graph-based approach is more accurate in terms of resolution of the computational graph and the corresponding deep learning architectural information since it only requires partial compilation.

7 References


[12] [Online]. https://github.com/davidfraser/pyan