**DARPA ASKE DCC – Milestone 6, 2019**

Authors: Akrotirianakis Ioannis, Amar Viswanathan Kannan, Fradkin Dmitriy, Roy Aditi, Canedo Arquimedes, Mohammad Al Faruque

**Contents**

[1. Introduction 3](#_Toc15483395)

[2. Summary and general architecture of DCC 3](#_Toc15483396)

[3. Multimodal knowledge graph extraction 3](#_Toc15483397)

[3.1 Graph alignment 7](#_Toc15483398)

[3.2 Work in Progress: 10](#_Toc15483399)

[4. Text2Graph 11](#_Toc15483400)

[4.1 Automatic annotation of text 11](#_Toc15483401)

[4.2 Improvements in the learning NER and RE models 13](#_Toc15483402)

[5. Image2Graph 13](#_Toc15483403)

[5.1 Overview 13](#_Toc15483404)

[5.2 Flow detection in the absence of arrows in DL diagrams: 14](#_Toc15483405)

[5.3 Interpretation of color code in arrows or in nodes and decoding legend: 15](#_Toc15483406)

[6. Code2Graph 16](#_Toc15483407)

[6.1 Overview 16](#_Toc15483408)

[6.2 Pre-Processing 17](#_Toc15483409)

[*6.2.1* *Pre-processing for the Lightweight Approach:* 17](#_Toc15483410)

[*6.2.2* *Pre-processing for the Computational Graph-based Approach:* 18](#_Toc15483411)

[*6.2.3* *Microservice for scraping the paperswithcode.com* 18](#_Toc15483412)

[6.3 Applying Knowledge Graph Embedding Algorithms 18](#_Toc15483413)

[6.4 Results and Future Works 20](#_Toc15483414)

[*6.4.1* *Statistics of the lightweight approach and computational graph-based approach* 20](#_Toc15483415)

[*6.4.2* *Results from Knowledge Graph Embedding (KGE) Methods* 21](#_Toc15483416)

[*6.4.3* *Future Works* 22](#_Toc15483417)

# Introduction

The purpose of this report is to provide an overview of the work that has been done during **June 1st** to **July 31st , 2019**. We present details for our multimodal (text, images, and code) knowledge graph extraction process and discuss the various updates in our three major modules: text2graph, image2graph, and code2graph.

# Summary and general architecture of DCC

We start by briefly reviewing the general architecture of our framework which is shown in Figure 1. The text2graph module generate a KG from text, where nodes are entities describing deep learning methods or tasks and the edges represent the relations between those entities. For the KG generated from images (*image2graph*), the entities are derived by the shapes used to describe the deep learning architecture and the text within them, whereas the relations are derived by the arrows connecting those shapes. Finally, the code2graph module generates a KG using the information extracted from the source code, where the nodes are actual functions used by TensorFlow, and the edges represent the way these functions are related to form the executable code. It should be noted that in order to construct each KG for the whole corpus, we first apply our models to individual papers and then integrate the entities and relations for all the documents and all modalities. We only focus on papers that use Python as the programming language and TensorFlow/Keras as the Deep Learning framework.

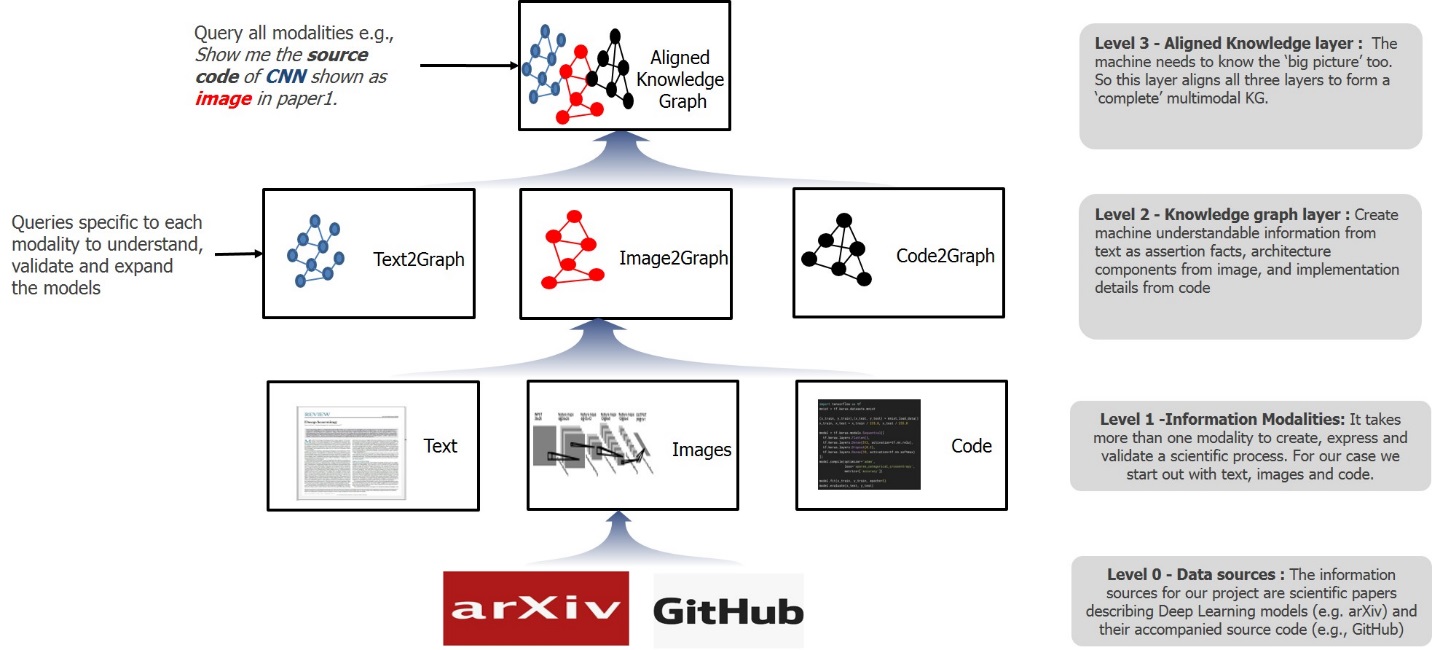


Figure 1: DCC architecture within the ASKE framework

# Multimodal knowledge graph extraction

In this section we present our initial results for the Knowledge Graph creation based on the multimodal information extraction mechanisms (text2graph, image2graph, and code2graph) we have developed.A knowledge graph encodes semantic information as uniquely identifiable entities and relationships between them. Thus, the entities are the nodes of the knowledge graph and the named edges are the relationships. Every unique node-edge-node combination is a Subject-Predicate-Object (SPO) triple. Each of these elements (i.e. the nodes and the edge) is uniquely represented by Internationalized Resource Identifier (IRI). This representation is the foundation of the Resource Description Framework (RDF), which is the standard for representing connected semantic content in knowledge graphs. These knowledge graphs are stored in triple stores and the information is queried from them using triple pattern queries. In addition to entity relationships the knowledge graphs also encode (as triples) the domain specific knowledge in the form of an ontology or schema. Popular large scale heterogeneous social knowledge graphs such as *YAGO[[1]](#footnote-2) and DBPedia[[2]](#footnote-3)* follow this standard to represent, curate and query for general knowledge. To the best of our knowledge, there is no such ontology for DL.

In Phase 2 we have focused on constructing a *multi modal linked repository for papers and source code related to DL algorithms and methods.* This will enhance and enable knowledge aggregation, and fine grained multimodal analysis of scientific publications in the deep learning domain. After carefully analyzing our dataset of DL publications[[3]](#footnote-4) and their reproducible source codes, we have begun the process of creating a unified knowledge graph for: (a) textual information; (b) images describing DL architectures; and (c) accompanying source code. The automatic extraction for each of these components were described in Milestone 3. While the automatic extraction, extracts components independently, it is disparate and can contain complementary information that would be better served if all the modalities are merged. To aid this process we started out with our first version of a curated multi modal ontology DeepSciKG. The current version of DeepSciKG includes the following components:

1. Vocabulary for representing the extracted components from the *text* of the scientific publication. In this case the vocabulary for this consists of the same terms that were used in the extraction.
2. Vocabulary for representing the published *metadata*. This includes the conference venue, conference title, published year, arxiv id, github repository
3. Vocabulary for representing the DL architecture *diagrams* extracted from the publications. This vocabulary is designed from the image extraction component and is used to describe the image as a knowledge graph and encompasses data flow as well.
4. Vocabulary for representing the extracted *source code* as a knowledge graph. In this initial version we focused on transforming the vocabulary of TensorFlow Python API [[4]](#footnote-5) to a schema. We mapped the Modules, Classes and functions to our ontology.

DeepSciKG is written in OWL 2 according to the design principles of clarity, coherence and extensibility. It consists of 189 Classes, 12 Object properties, 10 Data Properties, and 10 annotation properties and has been designed keeping in mind the extension possibilities to all the components. Figures 2 and 3 show snippets of the ontology schema. In Figure 2 one the RNN cell component is shown along with the various extensions to RNNs that TensorFlow allows. In Figure 3, we see the barebones skeletal structure of a multimodal entity. More specifically, the root of the ontology is Thing. Under this we have tf, which is the root of the TensorFlow schema. This root, tf, has subclasses which correspond to the modules present in the documentation. In addition to this, the ontology also encodes functions present in the documentation as part of the TensorFlow package. The root for this is the class Function. From our extracted graphs using the code2graph module, whenever we see instances of functions matching the TensorFlow vocabulary, we create instances of the respective functions, classes and modules, and add it to our knowledge graph. Furthermore, these instances have datatype and object type properties that link them to the paper from which they were derived and the code from which they were generated. The properties that are related to this are the object properties hasFunction, isPartOfPublication and the data properties hasPaperId, githubrepo and arxivId. Figure 2. shows the snippet of our DeepSciKG ontology schema for the TensorFlow API [[5]](#footnote-6). In this we can see the tensorflow hierarchy which is rnn\_cell *subClassOf* nn *subClassOf* Module *subClassOf* tf *subClassOf* Thing. rnn\_cell represents a recurrent neural network cell and the different variants of this implemented in TensorFlow are also shown (e.g., BasicLSTMCell, BasicRNNCell, MultiRNNCell, etc.)

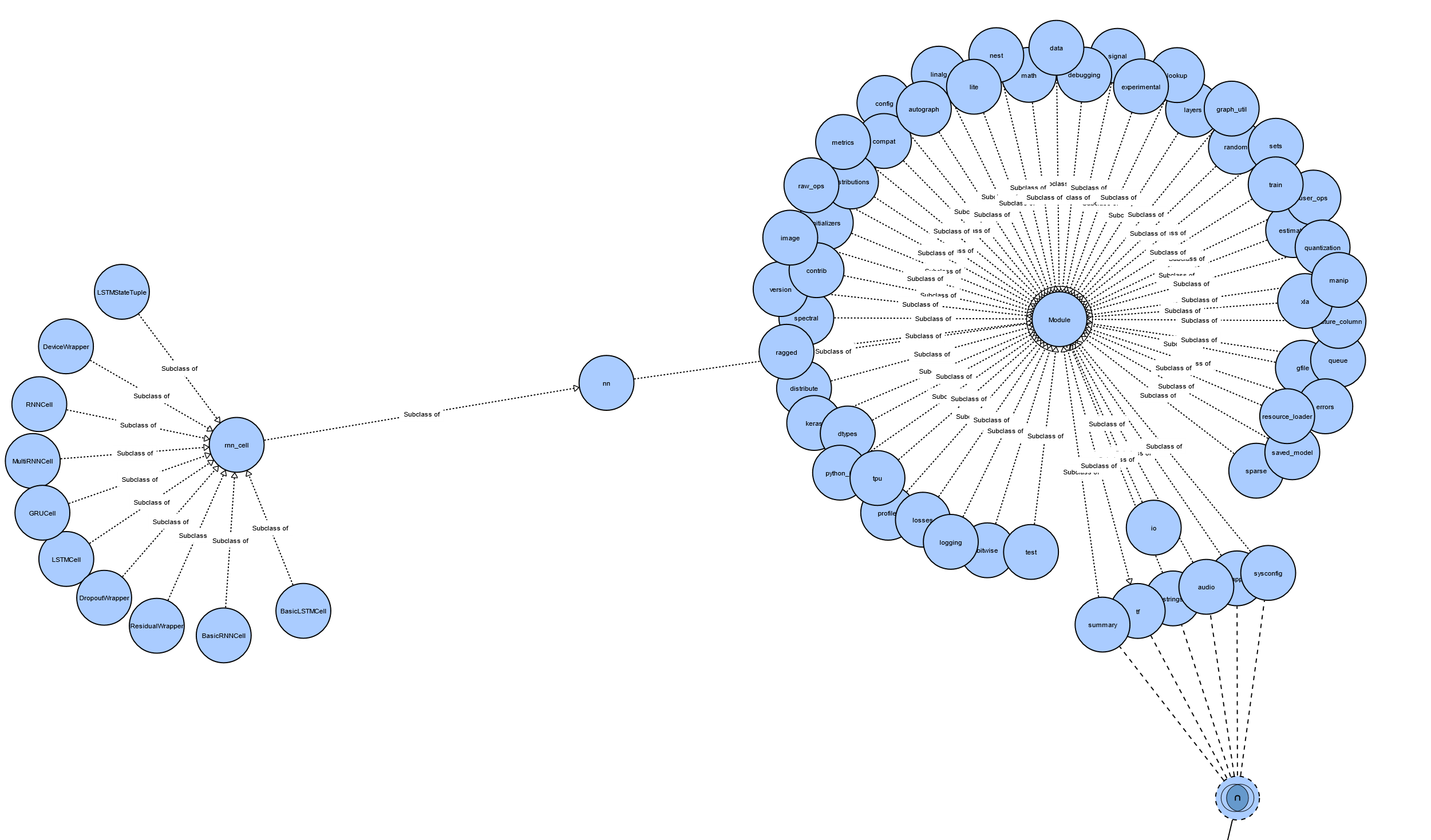


Figure 2: TensorFlow snipped component of the DeepSciKG ontology, focusing on RNN

In Figure 3, the schema for a multimodal entity is shown. An example multimodal entity is a convolutional neural network (CNN) which is present in the text as a CNN, and in the image as a set of CNN rectangular components. In the source code, the CNN is present as functions conv1d, conv2d, conv3d, etc. These are part of the tf.nn module. This is reflected in our ontology schema as well. Using the same CNN as an example, we see that CNN has data properties defined by and,compare,usedFor which come from text2graph. The properties followedBy and dataflow come from image2graph. Finally, this entity is linked to the previous TensorFlow functions by the hasFunction object type property.

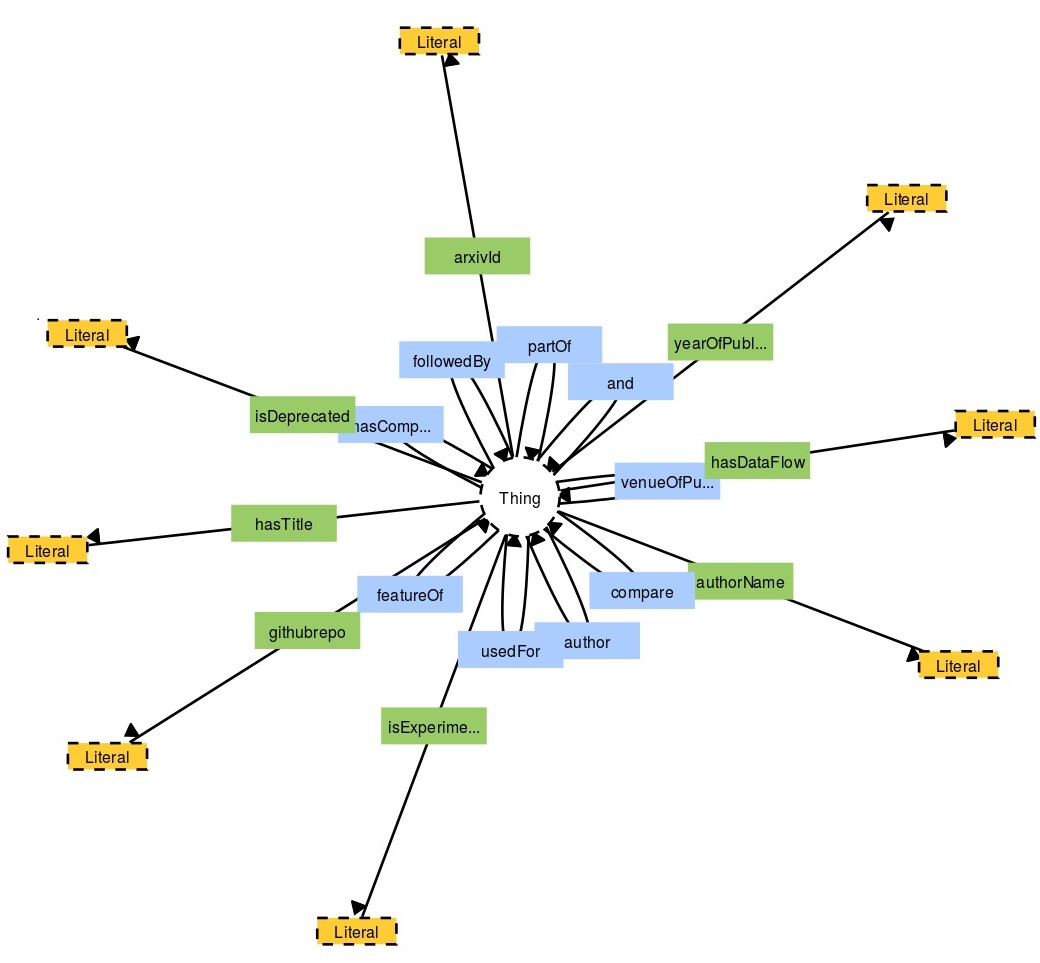


Figure 3: Multimodal entity component of the DeepSciKG ontology

## Graph alignment

In our first iteration of the alignment, the focus has been to figure out the right vocabularies to match to. In the case of the *image2graph*, the mapping has considered the textual content and uses a similarity metric to map it to the vocabulary in the ontology. Since all our vocabulary terms in the end match the actual code implementation, this makes creating a unified graph easier. In the case of *text2graph*, we look for similar entities from DBpedia and then map it to the right definition. This entry is also mapped to the respective code entry from the vocabulary. For *code2graph*, since the extracted graphs contain a lot of information, we only extract snippets from the call graph to match it to the right vocabulary element.

**Example:**

In Figure 4, we show the details of a snippet of the unified graph for the paper with title *Very Deep Convolutional Networks for Text Classification[[6]](#footnote-7)*, showing methods, publications and image components mapped to the code component.

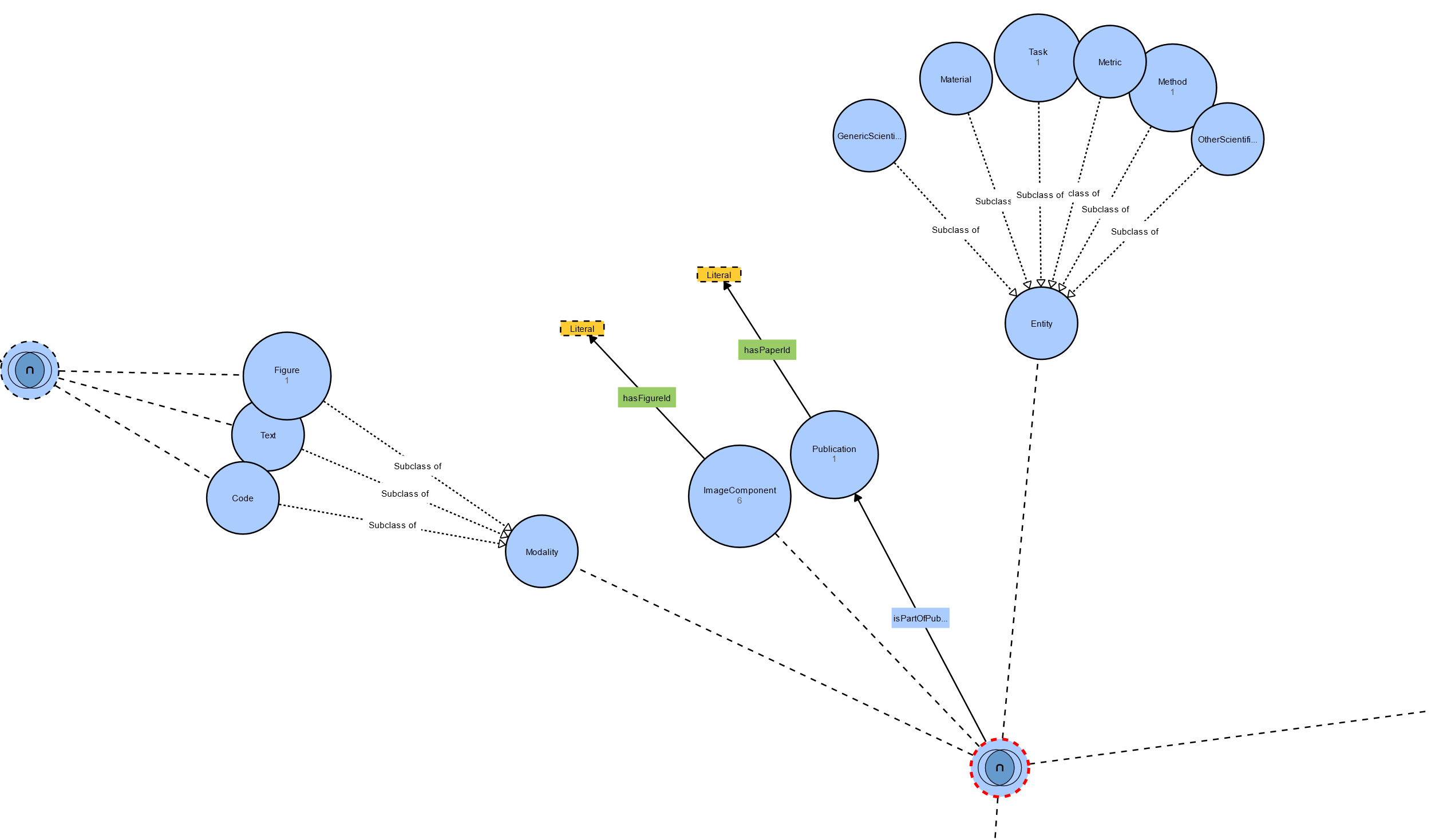


Figure 4: Snippet of a unified graph for all the three elements

While the text2graph component extracts only 2 triples, the image2graph component extracted the following triples:

"fig1606.01781v2-Figure2-1.png" "found in" "1606.01781v2.pdf"

"fig1606.01781v2-Figure2-1.png" "hasDataFlow" "bottom2top"

Component number 0 "has type": activation

Component number 0 "has description": ['relu']

Component number 1 "has type": norm

Component number 1 "has description": ['batch', 'norm', 'temporal']

Component number 2 "has type": conv

Component number 2 "has description": ['256', 'temp', '3']

Component number 3 "has type": activation

Component number 3 "has description": ['relu']

Component number 4 "has type": norm

Component number 4 "has description": ['temporal batch']

Component number 5 "has type": conv

Component number 5 "has description": ['256', 'temp', '3']

Node number comp5 "followed by" Node number comp4

Node number comp4 "followed by" Node number comp3

Node number comp3 "followed by" Node number comp2

Node number comp2 "followed by" Node number comp1

Node number comp1 "followed by" Node number comp0

From these above triples, the Components were matched to the right TensorFlow functions in the ontology. They are: relu, batch\_normalization, conv3d, batch\_normalization. The call graphs of the source code that were mapped are shown in Figures 5 and 6.

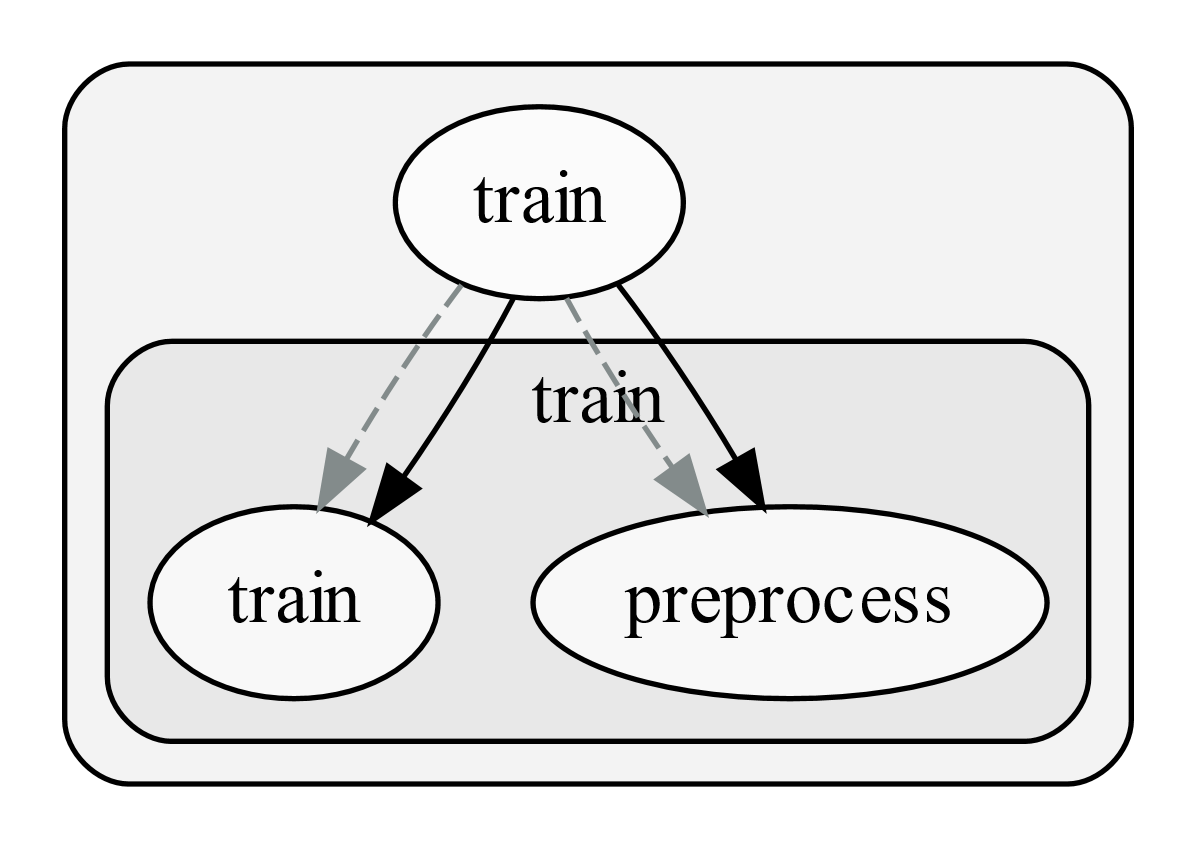


Figure 5: extracted call graph of the train.py file from vdcnn source code[[7]](#footnote-8)

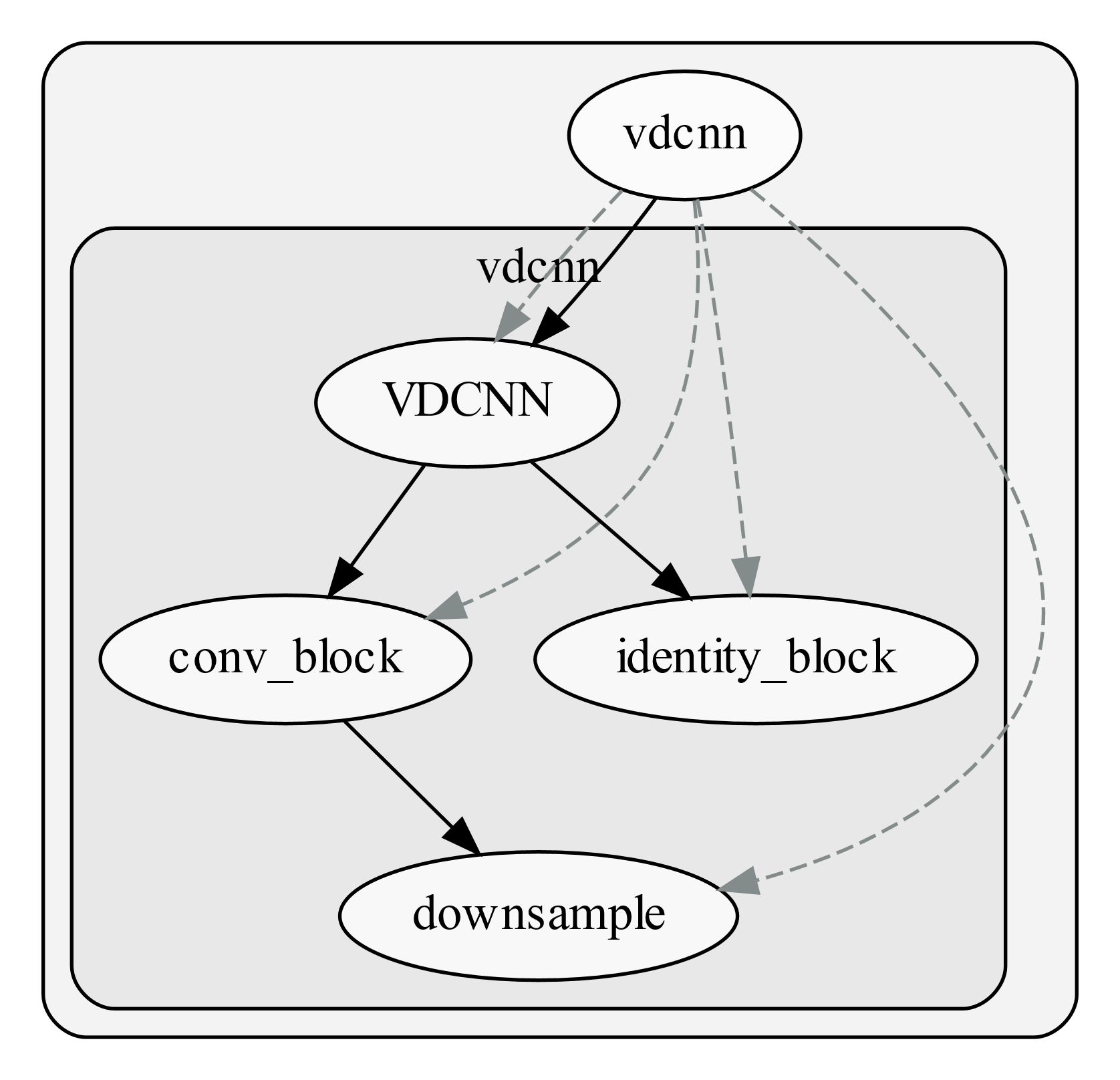


Figure 6: extracted call graph of the vdcnn.py source code

The training file calls the train function, which in turn calls the vdcnn class, which calls the convolutional block. This convolutional block has been mapped to the TensorFlow vocabulary element. This merged graph now can handle expressive queries such as “*publications which use convolutional neural networks with batch normalization”.*

## Work in Progress:

For the next milestone we will be working on the following tasks for the multimodal knowledge graph:

1. Convert all the extracted papers, images and code into the merged graph (OWL2 + RDF).
2. Automate the alignment process with more fine-grained details, particularly utilizing entity properties as features for alignment
3. Deploy the graph in a public triple store

# Text2Graph

The general architecture of the text2graph module remains the same as in Phase 1 and is shown in the Figure 7. In Phase 1, we extracted and the text from the papers and annotated (entities and relations) using the web-based tool Brat[[8]](#footnote-9). During the last two months we have been using the Computer Science Ontology[[9]](#footnote-10) to speed up and to some extend automate the annotation process. This effort will benefit the statistical modes we have developed for Named Entity Recognition (NER) and Relation Extraction (RE) and improve their prediction accuracy.

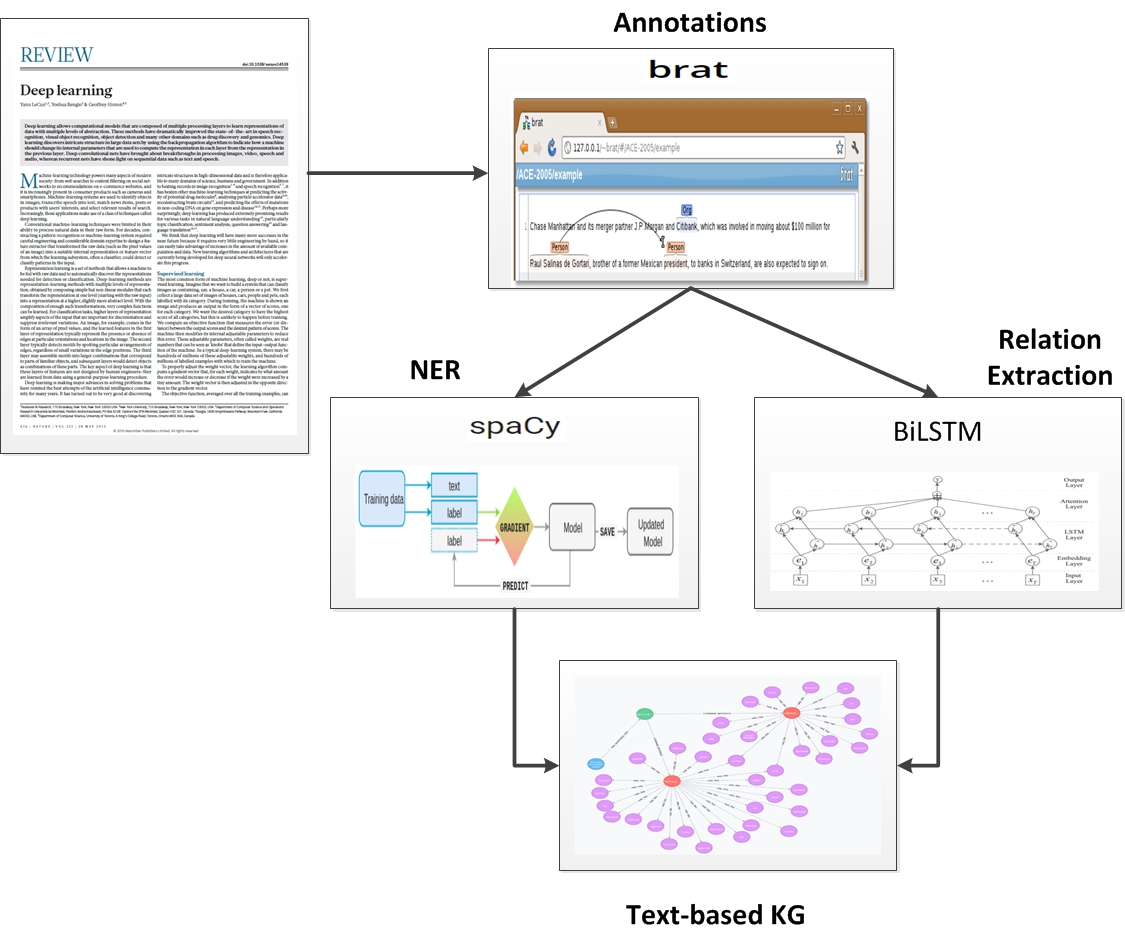


Figure 7: The text2graph pipeline

## Automatic annotation of text

In particular, our effort has focused on improving the efficiency of the statistical models for NER and RE, which are used to construct the text2graph component. We have focused on two main directions: (a.) implementing a process that allows us to automatically annotate new text, and (b.) improving the accuracy of the learning morels for NER and RE.

Good quality annotations play a very important role in the training process of our NER and RE models. So far, we have relied on human-based text annotations using the web-based tool Bart. This process resulted in annotating 100 papers describing Deep Learning (DL) methods, architectures and applications. In addition, we have created an *Ontology* which will provide (i.) a vocabulary of strings mapping to entity types of interest, and (ii.) a mapping from pairs of strings to relations of interest. The current dataset we have created consists of 1245 terms mapped to entities[[10]](#footnote-11) and 762 mappings from pairs of entities to relations[[11]](#footnote-12).

However, human-based annotations can be costly and difficult to scale up. We would also like to be able to increase our annotations for new papers in an easier and more cost-effective way. A promising data source we are currently utilizing is the Computer Science Ontology[[12]](#footnote-13) (CSO). CSO is a large-scale ontology of research areas consisting of about 16 million publications. The main root of CSO is the term *“Computer Science”* which contains all the major terms in the field of *Deep Learning*. There are a few additional roots (e.g., “Linguistics” and “Semantics”) which we expect to be useful for our project too. The CSO model consists of eight semantic relationships which will be used in our annotation scheme (e.g., ‘*sameAs*’ (synonym), ‘*superTopicOf*’ (inverse of isA) and ‘*contributesTo*’ (usedFor)).

We have developed a pipeline that can expand our vocabulary and set of relations by combining CSO and the human-based annotations we have generated from the original 100 deep learning papers. In particular, the process consists of the following steps:

1. Extract topics from CSO as strings and map them to Uniform Resource Identifiers[[13]](#footnote-14) (URIs)
2. Extract useful information from CSO corresponding to *usedFor* and *sameAs* relations[[14]](#footnote-15)
3. Find which CSO terms occur in our manually annotated data and generate sets of terms both from CSO and annotations that we can thus map to the same URI,
4. Create new URIs for those annotations that did not match any in CSO,
5. For all the URIs from steps 3 and 4 we create a mapping to our entity types (i.e., “Method”, “Data”, “Generic”) based on annotations,
6. Merge the relations from CSO and from our original annotations[[15]](#footnote-16)

The above process results in

* Mapping from strings to URIs with multiple strings mapping to the same URI: 15154 strings[[16]](#footnote-17)
* Mapping from URIs to Entity Types: 1173 URIs
* Mapping from pairs of URIs to Relations: 45925 pairs

We note that the above process may result in some conflicts between the CSO and the manually generated annotations. For example, the term “*language models*” was identified as a *Task* in CSO and as a *Method* in our manual annotations. We are currently working on improving the such conflicts.

To automatically annotate new pieces of text we will be using the data structures described above. Specifically, for a new piece of text that has not been annotated before, we follow the following steps[[17]](#footnote-18):

1. Find occurrences of all strings in our vocabulary in the text, with corresponding URIs,
2. Select those for which we have URI-to-entity mapping (this will be equivalent to named entity recognition),
3. For all pairs of strings matching entities check for relations in the URI pairs-to-relations map (this will provide an initial form of relation extraction).

## Improvements in the learning NER and RE models

The NER and RE learning models we have developed in the previous milestones use a number of parameters that may affect their performance. Finding the values of those parameters that result in increased efficiency of the overall learning algorithms is an important part of our effort. Currently we are using a grid search to determine the values of (i) the learning rate and (ii) the dropout rate. The learning rate controls how much to change the deep neural network models used for NER and RE in response to the estimated error produced each time the model weights are updated during the optimization process. Currently the learning rate is set to 0.00156. In addition the, we also investigated the effect of the dropout rate which helps improve the learned model. The main role of the dropout rate is to randomly ignore specific features of the input sentences, in order avoid memorizing the training dataset by the models and generalize better to unknown sentences. This means that during training a number of layers of the neural network are randomly ignored. We found that when we set the dropout to 0.33 we obtain the best generalization performance thus far[[18]](#footnote-19). Currently, for the entities model the f1-score and the recall are 78.89 and 80.0, respectively.

# Image2Graph

## Overview

The main architecture of the image2graph module is shown in Figure 8 and has not changed from that developed in Phase 1. Given the PDF of a deep learning research paper, the image2graph module consists of four major components: (1) extraction of all the figures from a research paper, (2) identification of those figures showing DL model diagram, (3) analysis of the diagrams depicting DL models, and (4) construction of a graph representing the information extracted from the DL diagram. During the last two months we have focused on improving the detection of the information flow, as well as interpreting different colors in nodes and arrows that are used in Deep Learning architectures. The results are summarized in the following sections.

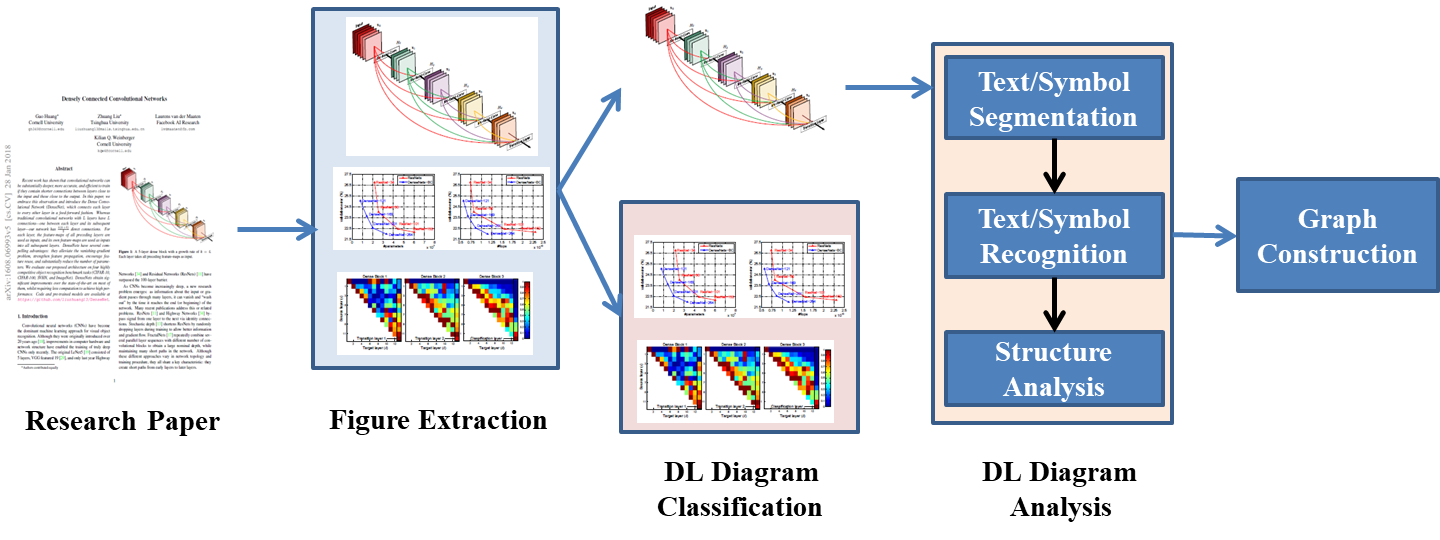
****

Figure 8 The image2graph pipeline

## Flow detection in the absence of arrows in DL diagrams:

There are certain deep learning diagrams where arrows are not present to describe flow of the deep learning model. It can be observed that in most of such deep learning diagrams, the flow is implicit in term of direction and possible node connections as shown in Figure 9.

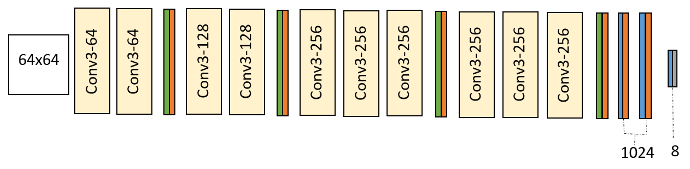
To parse such diagrams, we hypothesize that the flow of a diagram is either *left-to-right* or *top-to-bottom* in case of *absence of any arrow*. However, if the “input” component is present in the diagram, then that component is considered as the starting component. The diagrams having horizontal flow, flow direction is computed from left-to-right or right-to-left. Similarly, for vertical flow diagrams, flow is computed either *top-to-bottom* or *bottom-to-top* starting from the “input” component. In case “input” component is not present but “output” component is present, similar logic is used to detect flow direction considering “output” as the end component.

Figure 9: Deep Learning diagram without arrows showing flow direction

For the architecture shown in Figure 9, after structural analysis our proposed technique correctly identifies the flow which is shown in detail in Figure 10.

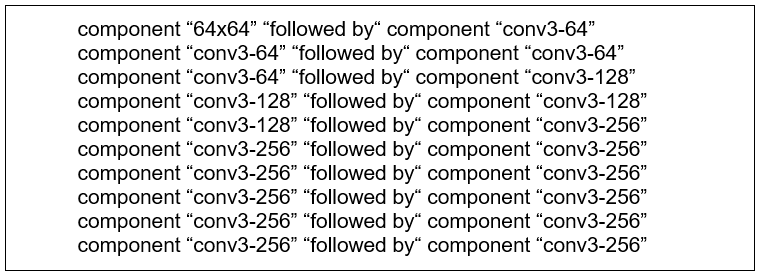


Figure 10: Information flow capturing from DL diagrams

## Interpretation of color code in arrows or in nodes and decoding legend:

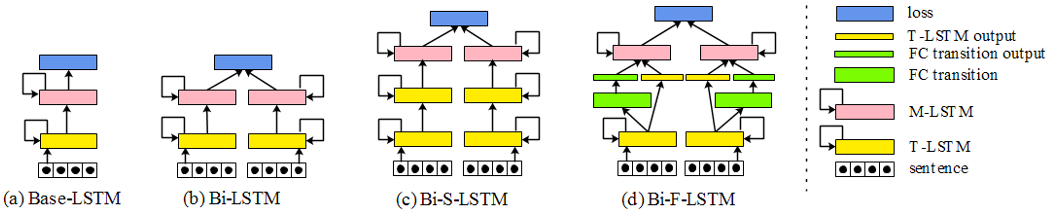
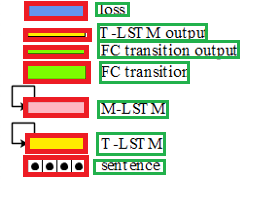


Figure 11: DL diagram with legends

Typically, the legend has entries consisting of (label, symbol) pairs, where the labels are the variable names (e.g., ‘loss’ in Figure 11) and the symbols give an example of its appearance. However, there is huge variation in the placement and format of legends across figures. Legend entries may either be arranged vertically, horizontally, or in a rectangle, and they may be found either outside the DL diagram area or anywhere inside. Further, the legend symbols may be placed either to the right or left of the legend labels and may have varying lengths with spaces.

To address this challenge, our legend extractor first analyzes the input image to identify whether any legend is present or not. We train a RCNN to detect legend regions if present in a DL diagram.

Once the legend region is detected, the legend labels and symbols are identified separately. For label identification, tesseract text recognizer is used to detect the text boxes in the legend region. Symbols are identified using the same algorithm applied for component detection from DL diagrams. Figure 12 shows the symbols and labels corresponding to Figure 11. Next, for localizing the symbols corresponding to the identified legend labels, we first determine their side (i.e., left or right of the text). Then, the candidate scores across all labels on each side (i.e., left or right) is compared to choose the final one with highest score. The selected candidate boxes are subsequently cropped to obtain the final symbol bounds.

Next challenge is associating the DL figure content with the legend entries. To address this problem, we propose to use Siamese network where each training example consists of a legend symbol patch and a DL node patch. If the legend symbol patch and DL node patch are of the same type whey will be classified as same class otherwise, they will be classified as different. Thus, we can identify all the nodes having the same symbol as a legend. Currently, we developed the Siamese network and tested it on existing open source reidentification dataset. We are creating the labeled dataset from DL architecture to train this Siamese network.

Figure 12: Legend with symbols and labels

# Code2Graph

## Overview

In code2graph, methodologies have been developed to extract the Resource Description Framework (RDF) representations from the code included in Deep Learning (DL) publications. During Phase 1 of the project, two main approaches have been studied: the Computational Graph-based Approach and the Lightweight Approach. In the *Computational Graph-based Approach*, we created a pipeline to extract and simplify the RDF graphs by executing the code (shown in green-colored boxes). On the other hand, the *Lightweight Approach* extracts the graphs by analyzing the abstract syntactic structure of the code (shown in blue-colored boxes).

The following sections describe the status of our work related to the code2graph in Milestone 6. The summaries of both extracting approaches were included in the Milestone 5 report. In Milestone 6, the main tasks associated with code2graph are:

1. Expanding the dataset that code2graph can tackle
2. Applying Knowledge Graph Embedding (KGE) Methods on the graphs generated by code2graph.

The updated overall pipeline is shown in Figure 13.

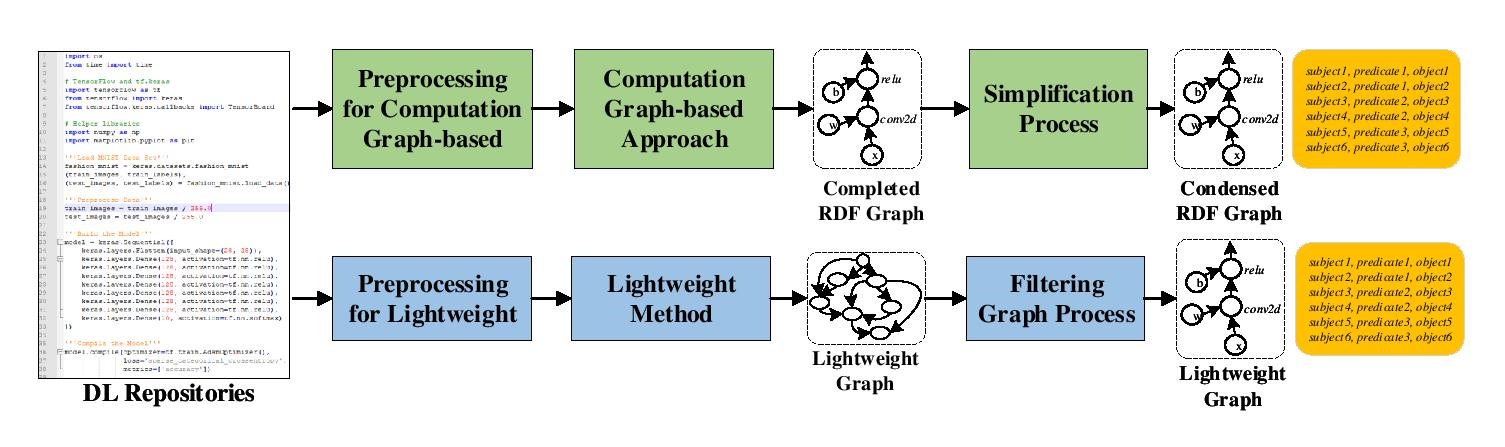


Figure 13: The *updated* pipeline for the code2graph

## Pre-Processing

In Milestone 6, we focused on improving the pre-processing component in order to scale up the dataset generated from both approaches. This enables us to perform some tasks in Knowledge Graph Completion from the Supergraph. During Phase 1 (Milestones 1 and 2), we learned that fully automating the preprocessing in the Computational Graph-based Approach (shown in green colored boxes of Figure 13) is quite a challenging process, since the developers may have different programming styles and assumptions in their Python code, thus making it hard for us to successfully find the ways to initiate the programs. Therefore, we have studied the Lightweight Approach as it might be an easier way to automate the process. Therefore, the preprocessing of the deep learning source code is now divided into two separate paths, with one specifically for the Computational Graph-based approach and the other for the Lightweight Approach.

### *Pre-processing for the Lightweight Approach:*

Although the Lightweight Approach doesn’t require the whole program to be executed, we still encounter issues while processing the TensorFlow papers in the dataset. Since the *ast* and *pyan* module we used in the pipeline are only applicable to Python 3, the Lightweight Approach will raise exceptions when running repositories that use Python 2, a mix of Python 2 and Python 3, or have indentation issues by themselves. To overcome these issues, we use two libraries: *2to3[[19]](#footnote-20)* and *autopep8[[20]](#footnote-21)*,which can translate the program files from Python 2 to Python 3 and make the indentation consistent throughout the files. The commands for achieving these are:

1. 2to3 -n -W [The directory path of the code]
2. autopep8 --**in**-place --aggressive -r [The directory path of the code]

### *Pre-processing for the Computational Graph-based Approach:*

Previously, we mentioned that using the Computational Graph-based Approach requires executing the programs until the construction of the computation graph is completed. The preprocessing steps include creating the required virtual environment, resolving the requirements for running the repositories, identifying the main program to execute, and injecting the code that assists in extracting the summary file. We have developed *automation scripts* for most of the steps. However, the parts that still require manual effort are identifying the main program to execute and correctly parameterizing the execution. We have tried finding the keywords “if \_\_name\_\_ == “\_\_main\_\_”: in the program or analyzing the readme files included in the repositories. However, some developers might just not include this information and might have very different programming conventions that are difficult to track via automation scripts. The best we can do, at the moment, to address this issue is to manually record it for each repository in the dataset, making the whole preprocessing pipeline *semi-automated*. In this manner, we then can reproduce the result for later use.

### *Microservice for scraping the paperswithcode.com*

The original dataset we acquired from Siemens contains 100 papers, 16 of which had repositories that adopt TensorFlow and Keras as the primary machine learning framework. Comparing to known Knowledge Base datasets such as FreeBase15K, FreeBase15K-237, Yago, WordNet18, etc.., the number of triples we can acquire from the 100-paper dataset is far from enough to perform KGE tasks. Therefore, we have revised the web-scraper and have made it a microservice “PWCScraper” that resides in the server and constantly crawls the latest TensorFlow papers from paperswithcode.com. The metadata that PWCScraper crawls from each webpage item (shown in Figure 14) include “title”, “abstract”, “paper link”, “code\_link”, “tags”, and the compressed source code from its master branch. The “tags” are the information associated with the tasks that the repository is capable of tackling and are labeled by the members in paperswithcode community.

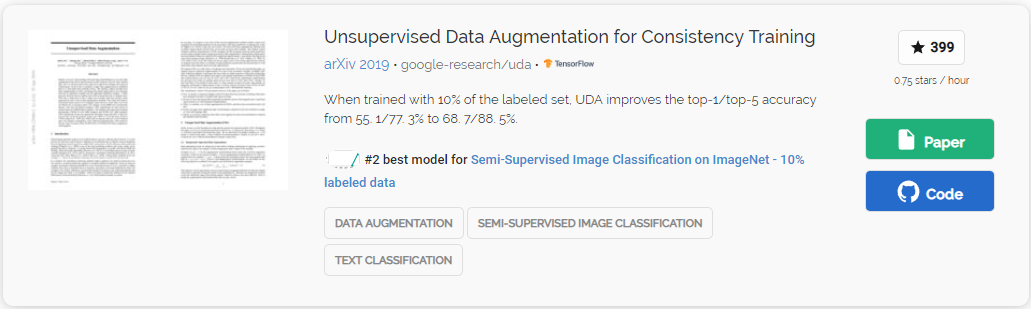


Figure 14: The snapshot of the item that PWCscraper crawls

## Applying Knowledge Graph Embedding Algorithms

Apart from expanding the dataset, we have also developed the pipeline of inference tasks on the graphs generated by code2graph methods, which are inherently heterogeneous graphs. For example, the Lightweight graph represents the static inter-relationships of the functions associated with the DL code as shown in Figure 15 and contains both hierarchical and sequential information. Each of the nodes (vertexes) in the Lightweight graph can be a “Python module”, a “Python function”, “a TensorFlow API function”, or a “literal value” indicating the attribute of a TensorFlow function. The links (edges) in Lightweight graphs can be ‘call’, ‘followed\_by’, or any specific relations related to attributes.

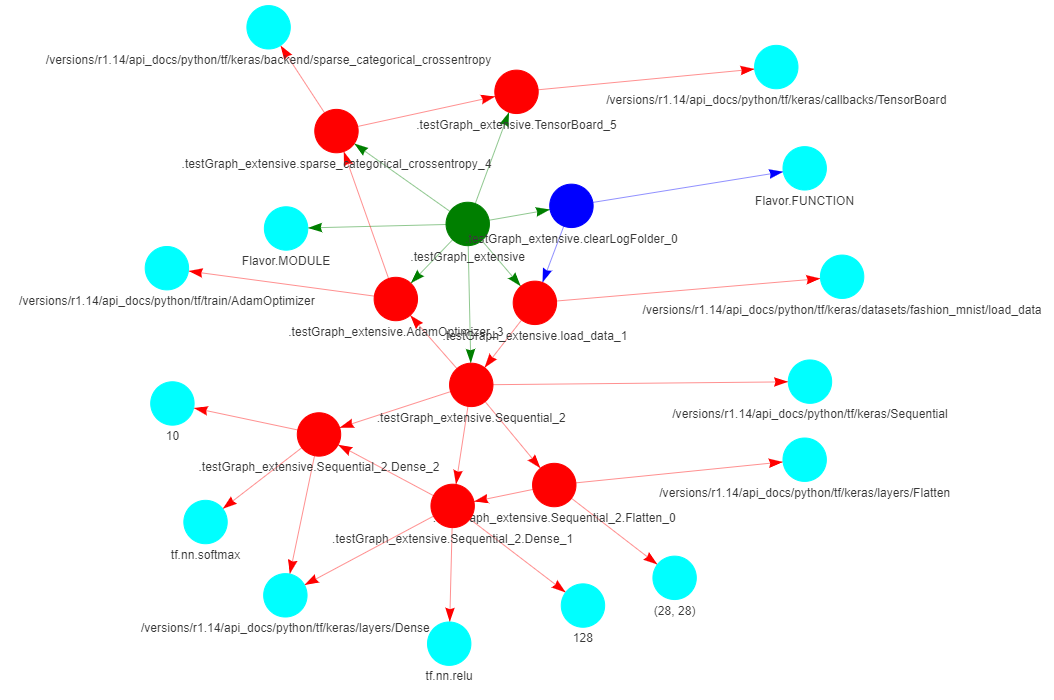


Figure 15: The output of the Lightweight approach

Considering the Lightweight Graphs as Knowledge Graphs, we then tried to integrate the KGE pipeline into code2graph. In the field of KGE, a Knowledge Graph contains a set of entities E and relations R between entities. The set of facts, **D+**, in the Knowledge Graph is represented in the form of triples (h, r, t), where h, t ∈ E are referred to as the head and the tail entities and r ∈ R is referred to as the relationship. The task of KGE is to find a function that learns the embeddings of entities and relations using low-dimensional vectors such that it preserves the architectural information. To accomplish this, one typical way is to enforce the learning of entities and relations to be compatible with the information in **D+**. There are several choices for representations, such as deterministic points, complex numbers, etc. By creating a set of unseen negative triplets, **D-**, from positive triplets, **D+**, the scoring function, fr(h,t), is then defined to reward the positive triples and penalize the negative triples. Eventually, an optimization algorithm such as Adam[[21]](#footnote-22) or SGD optimizer will minimize the cost function over training samples.

The evaluation tasks in KGE are either *predicting the missing entities in negative triples* (?, r, t) or (h, r, ?) or *predicting whether an unseen fact is true or not*. On the RDF graphs, we then can infer the missing entities or relations on the partially completed RDF graphs. We have integrated an open-source software repository called pykg2vec[[22]](#footnote-23), which contains nearly 20 KGE models and related evaluation tasks. The evaluation metrics include the rank of the answer in the predicted list (mean rank), the mean reciprocal rank (MRR), and the ratio of answers ranked top-k in the list (hit-k ratio). To integrate the existing code2graph pipeline, we modified the exporting mechanism of both graph extraction approaches (option 5 in graphlightweight.py and saveRDFtriples in graphHandler.py) such that it will export the text-format triple list.

## Results and Future Works

In this section, we will show some results and describe the next steps.

### *Statistics of the lightweight approach and computational graph-based approach*

Using PWCScraper, we have already collected more than 500 scientific papers with accompanied repositories from paperswithcode.com. Among those 500 papers, 130 papers adopt TensorFlow or Keras as the machine learning framework. With the pipeline of the Lightweight method, we have successfully extracted the RDF graphs for 60 papers. Since the computation graph-based approach requires manual effort to figure out how to execute the main programs, at this moment, we have only 10 RDF graphs successfully extracted. The statistic is shown in Table 1, with 10\_Lightweight representing the result of running the lightweight approach on 10 papers and 10\_Comp corresponding to the result of running the computational graph-based approach on the same 10 papers. 60\_Lightweight means running the lightweight approach on all the available TensorFlow repositories we have at hand.

In Table 1, we show that the dataset we acquired from the code2graph pipeline is comparable in size to known KGs such as Freebase15k, Wordnet18, Wordnet18\_rr, and Yago3\_10 in terms of the total number of triples. Therefore, it should be ready for giving some meaningful results of inferring missing entities or relations on the RDF graphs. Besides, another observation from Table 1 is that the Lightweight approach generates a larger set of unique entities compared to the ones generated from the Computation Graph-based approach. The reason is that the graphs generated from the computation graph-based approach have already been simplified using the rule-based method mentioned in the Phase 1 report, while the ones generated from the Lightweight approach haven’t been properly filtered or simplified. That is to say, the Lightweight graphs might contain information *irrelevant to the DL models* associated with the papers, because it extracts the graphs by analyzing the whole repository instead of its computational graphs. Therefore, in the next step, we will work on *filtering the Lightweight graphs*. In addition, we will continue to increase the number of repositories that our approach can process as 60 papers are still far from the amount necessary for further advanced applications such as code embedding or code classification. In the meantime, we will work with experts in Siemens on improving the RDF graphs so that they fulfill the common standards used in the team.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **# of train triplets** | **# of valid triplets** | **# of test triplets** | **# of unique entities** | **# of unique relations** |
| **Freebase15k** | 483142 | 59071 | 50000 | 14391 | 1345 |
| **Wordnet18** | 141442 | 5000 | 5000 | 40943 | 18 |
| **Wordnet18\_rr** | 86835 | 3134 | 3034 | 40943 | 11 |
| **Yago3\_10** | 1079040 | 5000 | 5000 | 123182 | 37 |
| **10\_Lightweight** | **43072** | **5385** | **5385** | **18987** | **42** |
| **10\_Comp** | **18016** | **2252** | **2252** | **9520** | **3** |
| **60\_Lightweight** | **253129** | **31642** | **31642** | **108974** | **98** |

**Table 1. The statistics of the known knowledge graphs and the graphs generated from code2graph.**

### *Results from Knowledge Graph Embedding (KGE) Methods*

For the dataset 10\_Lightweight, 10\_Comp and 60\_Lightweight, we have run two translational KGE methods and have acquired some preliminary results. The evaluating task we applied in the trial is the prediction of missing entities in the RDF graphs (?, r, t) or (h, r, ?). The algorithms we tried were TransE[[23]](#footnote-24) and TransH[[24]](#footnote-25). TransE is an energy-based model that represents the relationships as translations in the embedding space. Specifically, it assumes that if a fact (h, r, t) holds, then the embedding of the tail ‘t’ should be close to the embedding of head entity ‘h’ plus some vector that depends on the relationship ‘r’. TransH follows the same general principle as TransE. However, it differs from TransE since TransH introduces relation-specific hyperplanes. The entities are represented as vectors just like in TransE, however, the relation is modeled as a vector on its own hyperplane with a normal vector. The entities are then projected to the relation hyperplane to calculate the loss. The results are shown in Table 2, with all of them running the default hyperparameters (embedding dimension equal to 50) over 500 epochs. TransE and TransH have similar results for predicting the missing entities, however, current results are sub-optimal. We believe this is primarily because we currently adopt the default set of hyperparameters for training models so that the embedding dimension is insufficient to represent the features of entities and relations. Another possible reason is that those models are trained using the set of simple facts D+, while the multi-hop relation paths can reveal more architectural information in the graphs than the set of one-hop triples. In the next step, we will work on more qualitative and quantitative analysis as it helps us to gain insight on how to improve the RDF graph extracting pipeline. We will also apply more KGE methods that are available in pykg2vec. Besides, we will further study the KGE methods that consider path-related information and ontology information so that the sequential and hierarchical information is considered in finding the embeddings of entities and relations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| (TransE/TransH) | # of unique entities | Mean Rank | Hit-10 Ratio (%) | Hit-5 Ratio (%) |
| 10\_lightweight | 18987 | 2721.6/2899.136 | 16.9/14.0 | 11.0/9.5 |
| 10\_Comp | 9520 | 1672.27/1738.294 | 17.3/17.7 | 12.1/12.9 |
| 60\_lightweight | 108974 | 15436.2/16185.5 | 7.4/8.3 | 5.0/6.0 |

**Table 2. The results of TransE and TransH on 10\_Lightweight, 10\_Comp and 60\_Lighweight.**

### *Future Works*

Our focus in the next months will be in the following two directions

**1) Continue on Expanding and Filtering the RDF graphs**

In order to start training the inference algorithms for RDF graph completion, we have focused on both manual and automated dataset extraction of the code RDFs. We will continue working on automating the RDF extraction pipeline as far as possible from the computational and lightweight methods. For the computational graph, we have continued the task of mapping the nodes with the TensorFlow APIs. We are exploring both manual and semi-automated annotation options for RDF extraction and mapping to the deep learning architecture ontology. The mapping to the ontology will be crucial for creating the knowledge base and effectively aligning it with other RDF graphs.Both the lightweight and the computational graph methods generate a large amount of metadata (either related to the underlying code implementation or the coding structure utilized by the authors). Hence, we have been focusing our work on creating rule-based methodologies to filter the RDF graphs. For the Lightweight method (which extracts the RDF from the static call graph analysis), we have identified and increased the number of nodes that can be filtered out from the call trees. These nodes are based on the structure of the Python APIs (both for TensorFlow and Keras). We have extracted these APIs from the webpage of TensorFlow while creating an ontology. For the computational graphs, we have increased the list of node filters that can detect the C/C++ runtime metadata from the deep learning architecture information.

**2) Fine-tune the Hyperparameters and Apply more KGE Methods**

From 4.2, we have already built a pipeline of KGE methods that can process the RDF graphs that we generated from both approaches and subsequently train the embeddings for all entities and relations in the graphs, performing missing entity prediction in the level of triples. The next step will be vastly utilizing the pipeline to fine-tune the results by either finding the golden set of hyperparameters or improving the quality of our datasets. We will also implement different Knowledge Graph Embedding methods and compare the results between them. Besides this, we will also be studying more KGE methods that consider multi-hop relation paths or entity types so that the performance of predicting the missing entities on the RDF graphs can be further improved. In addition, we will explore the auto-encoder architecture for completing the head or tail entities when either is missing, predict relations given the two entities in the RDF triple, and create a sequential model to capture the RDF structure beyond first-order logic. This embedding will eventually aid in improving the code knowledge graph inferred by the team.

1. https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/ [↑](#footnote-ref-2)
2. https://wiki.dbpedia.org/ [↑](#footnote-ref-3)
3. Consisting of 100 papers [↑](#footnote-ref-4)
4. <https://www.tensorflow.org/api_docs/python/tf#classes> [↑](#footnote-ref-5)
5. <https://www.tensorflow.org/api_docs/python/tf#classes> [↑](#footnote-ref-6)
6. A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, “Very Deep Convolutional Networks for Text Classification”, In Proc. Of the15th Conference of the European Chapter of the Association for Computational Linguistics, Valencia, Spain, 2017; <https://arxiv.org/pdf/1606.01781.pdf> [↑](#footnote-ref-7)
7. https://github.com/zonetrooper32/VDCNN [↑](#footnote-ref-8)
8. http://rat.nlplab.org [↑](#footnote-ref-9)
9. https://cso.kmi.open.ac.uk/home [↑](#footnote-ref-10)
10. conflicts, i.e., same string mapping to different entities, are resolved arbitrarily at this point. [↑](#footnote-ref-11)
11. conflicts are resolved arbitrarily. [↑](#footnote-ref-12)
12. http://skm.kmi.open.ac.uk/cso/ [↑](#footnote-ref-13)
13. A Uniform Resource Identifier is a string of characters that is about naming, identifying, addressing and defining resources (https://en.wikipedia.org/wiki/Uniform\_Resource\_Identifier) [↑](#footnote-ref-14)
14. Steps 1 and 2 are performed by the program with name collect\_cso.py [↑](#footnote-ref-15)
15. Steps 3-6 are performed by the program with name: merge\_cso\_brat.py [↑](#footnote-ref-16)
16. A large number of those may not be relevant for our context of deep learning papers [↑](#footnote-ref-17)
17. These steps are implemented by the program with name auto\_annotate.py [↑](#footnote-ref-18)
18. This means that each feature has a 1/3 likelihood of being dropped from the network. [↑](#footnote-ref-19)
19. <https://docs.python.org/2/library/2to3.html> [↑](#footnote-ref-20)
20. <https://github.com/hhatto/autopep8> [↑](#footnote-ref-21)
21. D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization”, https://arxiv.org/abs/1412.6980 [↑](#footnote-ref-22)
22. <https://arxiv.org/abs/1906.04239> [↑](#footnote-ref-23)
23. <https://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf> [↑](#footnote-ref-24)
24. <https://pdfs.semanticscholar.org/2a3f/862199883ceff5e3c74126f0c80770653e05.pdf> [↑](#footnote-ref-25)