**DARPA ASKE DCC – Milestone 5, 2019**

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

**Contents**

[1. Introduction 2](#_Toc10214547)

[2. Summary and general architecture of DCC 2](#_Toc10214548)

[3. Text2Graph 3](#_Toc10214549)

[3.1 Overview 3](#_Toc10214550)

[3.2 Updates and next steps: 5](#_Toc10214551)

[4. Image2Graph 6](#_Toc10214552)

[4.1 Overview 6](#_Toc10214553)

[4.2 Updates and next steps: 8](#_Toc10214554)

[5. Code2Graph 12](#_Toc10214555)

[5.1 Overview 12](#_Toc10214556)

[5.2 Updates and next steps 14](#_Toc10214557)

# Introduction

The purpose of this report is to provide an overview of the work that has been done so far focusing on the lessons we have learned during Phase 1, describe the various updates we have made in our three major modules (text2graph, image2graph, code2graph) during the period May 1st and May 31st 2019, and report on our (future) work during Phase 2 which we are currently working on.

# Summary and general architecture of DCC

The area of Deep Learning (DL) has witnessed increasingly large number of publications and software implementations. The field of DL today, from ideation, dissemination and implementation encompasses a wide range of modalities. In our view, among the most important modalities are *textual content* for ideation, *visual content* for easy presentation of complex architectures, and finally *source code implementation* for result generation[[1]](#footnote-2). It is estimated that there is one Machine Learning paper every 2 minutes submitted to the arXiv database[[2]](#footnote-3). These papers use different programming languages and DL frameworks to implement the proposed algorithms[[3]](#footnote-4). It is a great challenge for researchers and developers to keep up with this rapid growth and be able to effectively use the results in their domain. We thus view at Deep Learning as a scientific sub-field that needs large scale automatic machine curation. This would make it easy for practitioners of various scientific fields (not necessarily within Computer Science) to easily query for model specific information, be abreast with the latest developments in state-of-the-art DL techniques and apply them in their own domain of expertise.

To address this challenge, we are currently developing a novel framework that can extract and automatically curate model specific information derived from DL papers and their implementations. We choose knowledge graphs (KG) as our representation for the extracted information. KGs provide the necessary *semantics* to represent domain specific DL information. In addition, they also come with the necessary machinery for large scale expressive querying. Thus, for each of the three main modalities we build KGs.

The general architecture of our system is shown in Figure 1. For the KG generated from text (*text2graph*), nodes are entities describing deep learning methods or tasks, whereas the edges represent the relations between those entities[[4]](#footnote-5). 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, for the KG created by the information extracted from the source code (*code2graph*) 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. Also we focus on papers that use Python as the programming language and TensorFlow/Keras as the DL framework.

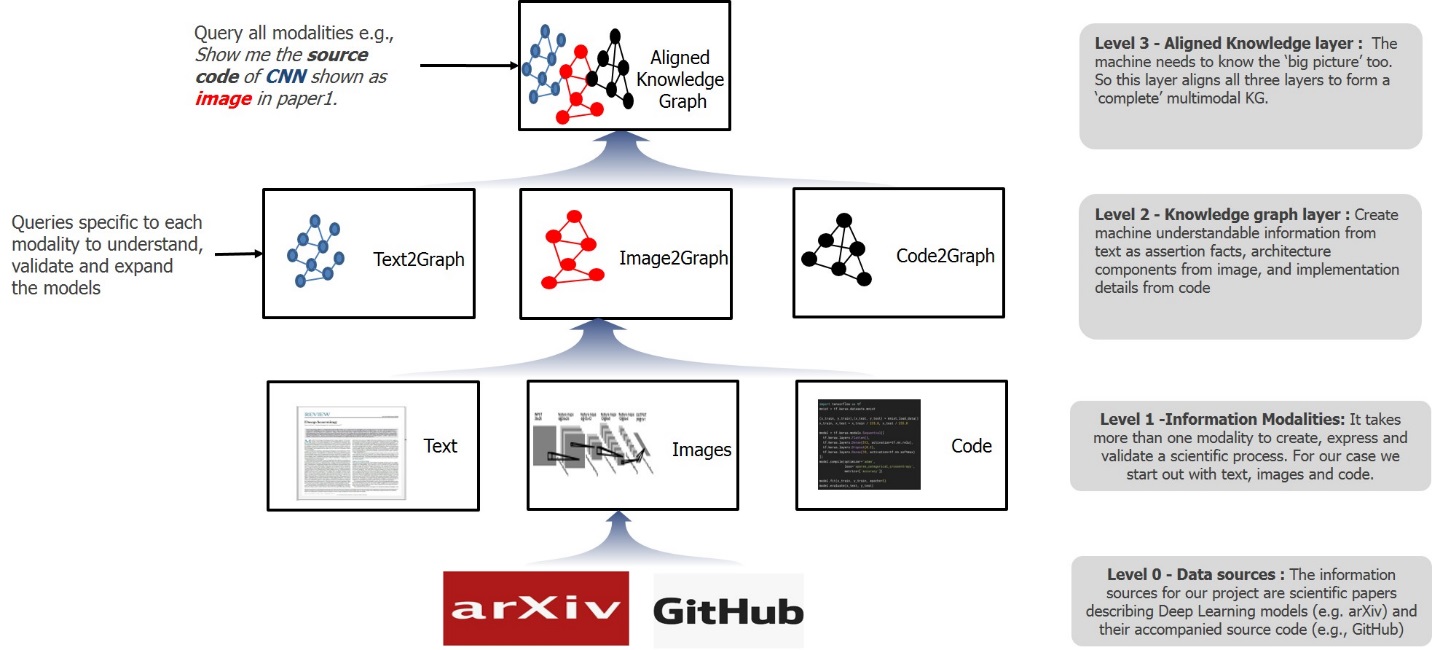


Figure 1: DCC architecture within the ASKE framework

# Text2Graph

## Overview

During Phase 1 we developed an end-to-end approach that extracts entities and relationships from the text obtained from scientific articles (in PDF format) describing deep learning (DL) models, algorithms, architectures and applications. Our approach is also able to generate a knowledge graph (KG) that integrates the extracted entities and relations from each article. To achieve our goal we have followed three main steps: (1) annotated the DL papers, (2) develop machine learning models that can predict entities and relations in DL articles that are not annotated, and (3) use the entities and the relations to construct the KG.

The general architecture of the text2graph module is shown in Figure 2. In Phase 1, we extracted and the text from the papers and annotated (entities and relations) using the web-based tool Brat[[5]](#footnote-6). As we will describe in the following section, during Phase 2 we will be using the Computer Science Ontology[[6]](#footnote-7) to speed up the annotation process. Next, the annotated text is used as input to train statistical models for Named Entity Recognition (NER) and Relation Extraction (RE). These models are used to predict entities and relations in new text that has not been annotated before. The predicted entities and relation are used to construct the knowledge graph of a DL paper. We have defined the following entities and relations:

***Entities*:** (i) **Task** - e.g., information extraction, forecasting, image analysis, (ii) **Method** - e.g., Neural Network, Attention, CRF, CNN, RNN, (iii) **Evaluation metric** - e.g., F1, Precision, Recall, ROC curve, (iv) **Material** - e.g., data, datasets, corpus, (v) **Other scientific terms** - e.g., dbpedia, Wikipedia, CoNLL, and (vi) **Generic** - e.g., model, approach, algorithm

***Relations*:** (i) **Used for – e.g.,** B is used for A, B models A, (ii) **Feature of -** B belongs to A B is a feature of A, (iii) **Part of –** e.g., Our system includes models A and B, (iv) **Compare** - Comparing two works, (v) **Conjunction** - Symmetric relation, (vi) **isA** – e.g., DNN is a type of artificial neural network, and (vii) **sameAs** : ex. NMT, otherwise known as neural machine translation.

For named entity recognition we have used the spaCy[[7]](#footnote-8) library, mainly due to the following three main advantages: (a) it is considered one of the fastest libraries for large-scale information extraction, (b) it provides very efficient machine learning algorithms for various NLP tasks, and (c) it incorporates easily with major deep learning libraries such as TensorFlow. SpaCy also enables the addition of arbitrary new classes to the NER model. This feature allowed us to easily integrate our six new entities defined above.

For relation extraction (RE) we developed a *Bidirectional LSTM* model along with a *neural attention mechanism* to capture the relationships. The main components of this model are: (i) Input layer which consists of the sentences we are considering, (ii) Embedding layer, which maps each sentence to a lower dimensional vector, (iii) LSTM layer, which implements a bidirectional LSTM that is able to take into account a large amount of context on both sides of a word and eliminate the problem of limited context that applies to any other DNN model, and (iv) Attention layer for improving the focus on words that have decisive effect on the classification of the relations and capturing the most important semantic information in a sentence.

Finally, we use the extracted entities and relations to build a knowledge graph based from the collection of DL documents. The KG is generated by triple statement as a {*subject, predicate, object*}, where the *subject* and *object* are extracted by our Named Entity Recognition model and the *predicate* is specified by the Relation Extraction component. Thus, we can view the triple statement as a {*entity1, relationship, entity2*}.

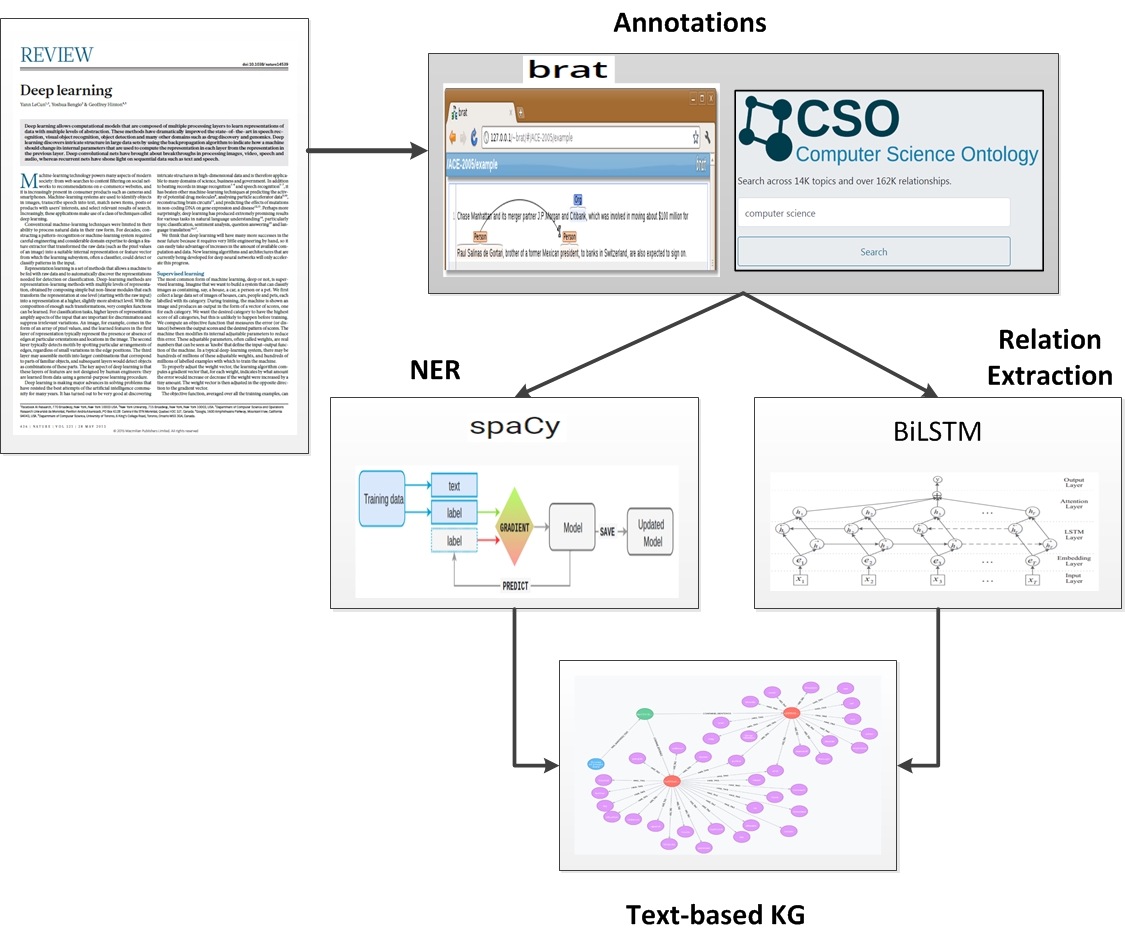


Figure 2: Text2Graph pipeline and architecture

## Updates and next steps:

During Phase 1 we had mostly generated annotations manually using the web-based tool Brat. This effort was sufficient and allowed us to develop our statistical models for NER and RE. However, annotating large documents manually is a very time consuming and expensive effort. Our goal in Phase 2 is to scale up our approach to full papers[[8]](#footnote-9) and increase the number of papers we consider. In addition, we want to have a larger coverage of relations of interest, while reducing reliance on our manually generated annotations. We have identified the following approaches to address these challenges and accomplishing our objectives:

1. To automate the generation of reliable labels, we are currently leveraging existing ontologies in order to detect known entities and relations in the text of a DL paper. These labels will be then used to train our statistical models, improving their accuracy, and expand the knowledge graphs generated for specific papers. We are leveraging the Computer Science Ontology[[9]](#footnote-10) (CSO), which 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 plan to integrate CSO into our KG and will consider occurrences of entities and relations from CSO in text as valid annotations.
2. We are currently working towards incorporating advanced features and modifications in the training our NER and RE models. In particular, we are focusing on (i) the use of general Part-Of-Speech (POS) taggers to get another sequence of features and enhance the accuracy of the models, (ii) use of general dependency parsers, and (iii) use of n-gram features.
3. *Distant supervision:* We also plan to leverage existing general purpose RE models, such as those trained on Wikipedia and SEMEVAL tasks to identify candidate relations in our datasets. The results will be checked against our annotations and combined with other methods using the SNORKEL[[10]](#footnote-11) approach.
4. We have so far limited our RE methods to relations occurring within single sentences. Performing RE across sentences is complicated by possible large distances between entities as well as the need for coreference resolution. Recently, the NLP community has turned its attention to RE in entity pairs that span multiple sentences. We will explore the use of inter-sentential dependency-based neural networks[[11]](#footnote-12) which are currently under developed in Siemens. The main aim is to develop deep neural network architectures that can detect relations between entity pairs that span different (consecutive) sentences. Two main variations will be developed, (i) neural networks that model the shortest dependency path spanning sentence boundary, and (ii) neural networks that model augmented dependency paths. The backbone of the new models will be the Bidirectional LSTM we have developed in Phase 1.

# Image2Graph

## Overview

In the image2graph module we have developed an end-to-end framework that automatically localizes all figures from a research paper, classifies them, extracts the content of the DL architecture figures and represents it in the form of a graph. Given the PDF of a deep learning research paper, the image2graph module consists of four major components, as shown in Figure 3: (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. A brief summary of each of these components follows.

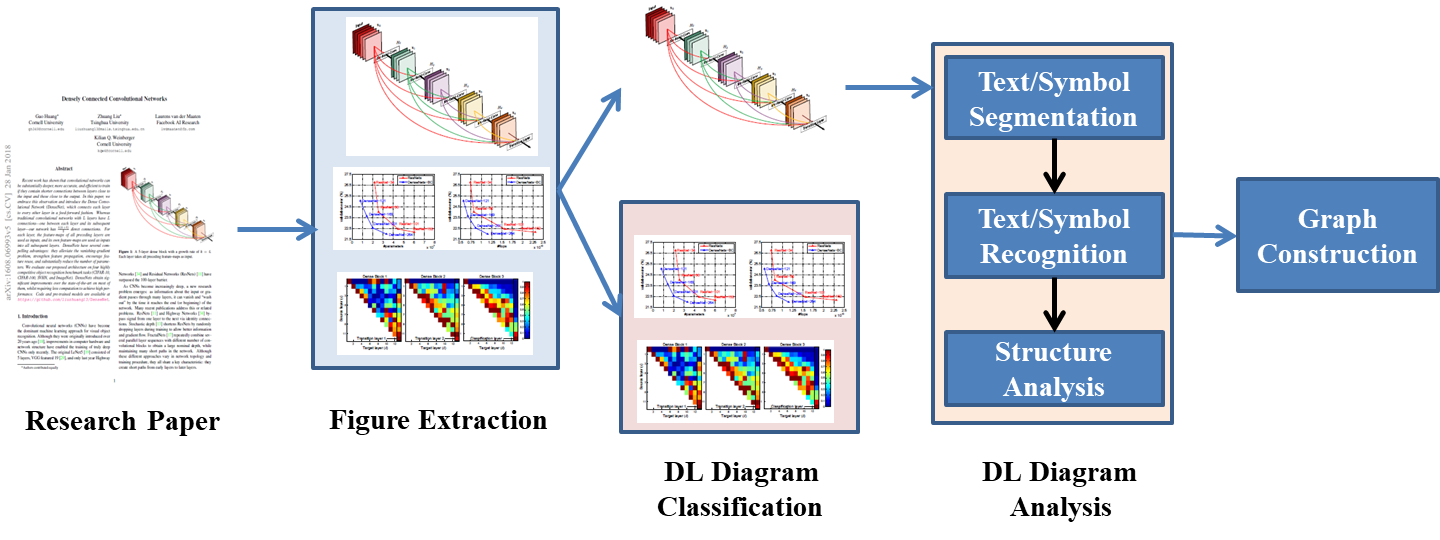
****

Figure 3 The image2graph pipeline

**1. Extract figures from research paper:**

We used the PDFFigures[[12]](#footnote-13) libraty for automatically extracting a list of figures from a research paper. We downloaded 1000 papers from arXiv.org using “deep learning” as the input query which provided 8310 figures.

**2. Figure classification**:

Figure classification is performed in two steps. Initially, a binary classifier has been developed that identifies and retrieves the diagrams/images describing DL models and/or architectures in the paper[[13]](#footnote-14). Next, we categorized the model diagrams into 6 broad types depending on the representation style (2D Box, Stacked2D Box, 3D Box, Matrix Box, Neurons plot, Pipeline plot). We trained a fine-grained 6-class classifier to categorize the figure into one of the six broad categories. In Phase 1 we focused on 2D Box DL diagrams for further analysis, while similar approach will be generalized to the other types in Phase 2, as it will be described later in this section.

**3. DL architecture diagram analysis:**

DL architecture analysis involves the following basic steps: (1) identification of the building blocks, (2) identification of the text present around those blocks, (3) identification of the edges connecting the nodes, and finally (4) determination of the spatial and logical relations among the nodes, edges and text candidates in order to generate the final interpretation of the DL architecture.

**4. Graph construction:** After detecting deep learning architecture and the flow of the computations within it, a computational graph is created with the following relationships describing each node or pair of nodes: “is type”, “has description”, “connected to”, “followed by”, “has input”, and “has output”.

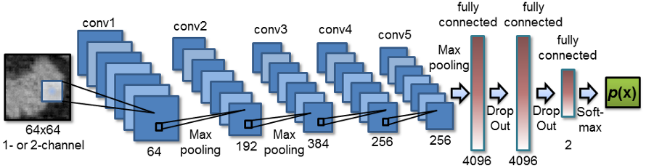
## Updates and next steps:

In phase 2 we currently focusing on the following topics.

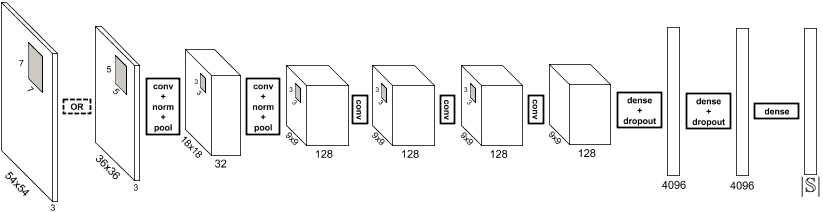
**1. Extending the node detection algorithm beyond 2D DL diagrams**

We aim to make the node detection algorithm more general such that nodes from other types of DL diagrams can also be detected. Specifically, we aim to handle the following types of DL model descriptions: (i) stacked 2D box plot, (ii) 3D box and (iii) matrix box plot. Examples of these DL models and the way they are frequently represented are shown in Figure 4. One approach we are currently focusing on is to automatically detect the objects/symbols in an image and the relationship between these objects/symbols. Recently, a deep learning-based technique named *faster R-CNN[[14]](#footnote-15)* has been applied for this purpose. We plan to further explore this approach with attention-guided learning to remove bias in training data and re-focus the network’s attention on the right and consistent patterns encountered in DL models. Our aim is to provide robustness and consistency in detecting the right building blocks of DL architectures and the information they convey (e.g., there are different ways to represent a CNN architecture that are used in DL papers).

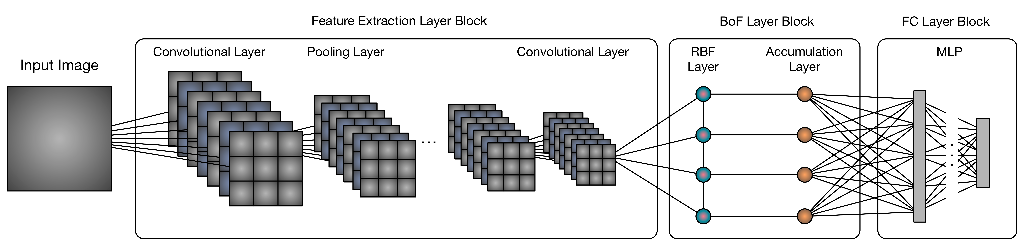
Top-down attention maps can be seen as one form of interpretation of neural networks. Based on our *Guided Attention Inference Network*[[15]](#footnote-16) (GAIN) framework, regularization on the attention maps will be used to guide the learning of detecting the DL building blocks more efficiently and effectively.



(a)



(b)



(c)

Figure : DL design diagrams: (a) stacked 2D box, (b) 3D box, (c) matrix box

**2. Arrow detection algorithm for various types of arrows:**

Our baseline algorithm currently cannot detect specialized arrows present in the DL diagram as shown in Figure 5. We have planned to handle this issue in Phase 2. A substantial work has been done on detection of *arrows in road scenes* using onboard cameras. Existing approaches perform road arrow detection in on-board camera images using inverse perspective mapping[[16]](#footnote-17) (IPM) of input images, using connected component analysis[[17]](#footnote-18), applying wavelets and matching the extracted features using curve/spline fitting models, geometrical pattern matching, and deep learning[[18]](#footnote-19).

We will handle this challenging task of arrow detection in two steps: a) localizing the head of the arrow, and (b) searching for the shaft of the arrow staring from the head. One possible approach could be to define the Regions of interests based on some preliminary segmentation knowledge in order to reduce the search space. Then head prototypes encoded as arc splines can be used for the comparison with the extracted contours of object candidates, which enables both detection and classification.

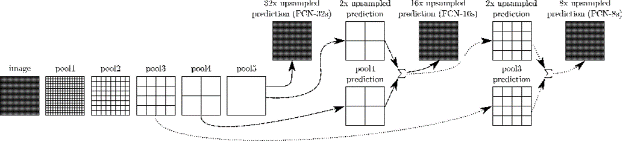


Figure : Detecting dotted curved arrows

**3. Improve the DL dictionary with inclusion of more keywords:**

We plan to increase the size of the current DL dictionary to make it more comprehensive.

**4. Structural analysis of DL diagrams to address the following challenges:**

Our focus will be in the following areas:

*Flow detection in the absence of arrows in DL diagrams:* It can be observed that in most of the DL diagrams where the arrows are not present to show the flow of the DL model, the flow is implicit in term of direction and possible node connections as shown in Figure 6.

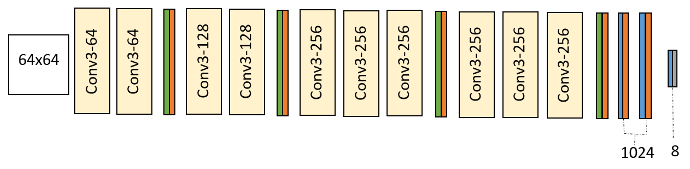


Figure : DL diagram without arrows to indicate the flow

We hypothesize that the flow of diagram is either left-to-right or top-to-bottom in case of absence of any arrow[[19]](#footnote-20). Additionally, we propose to employ a grammar defining the list of possible next layers for a given current layer. Once we detect all the nodes with their layer names, the connectivity among them is detected based on these two heuristics by filtering out all possible nodes that do not follow the two rules.

*Complex node connection with overlapping arrows:* This problem will be handled together with the arrow detection problem mentioned earlier. After detection of the arrow head, all possible candidate arrow shafts will be searched. Next, these candidate connections will be validated based on the above-mentioned list of possible next layers for a given layer in order to create valid deep learning design models.

*Interpretation of color code in arrows or in nodes and decoding legend:* Typically, the legend has entries consisting of (label, symbol) pairs, where the labels are the variable names (e.g., ‘loss’ in Figure 7) and the symbols give an example of its appearance. The challenge is that 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 these challenges, our legend extractor will identify the legend labels first, and then locate their corresponding symbols. The problem can be formulated as the legend label identification as a text classification problem, i.e., *given a text box within the figure, is it a legend label or not?* For localizing the symbols corresponding to the identified legend labels, we first need to determine their side (i.e., left or right of the text). Next, the candidate scores across all labels on each side (i.e., left or right) can be compared and then we will choose the final one with the highest score. The selected candidate boxes are subsequently cropped to obtain the final symbol bounds.

The next challenge is associating the DL figure content with the legend entries. To address this problem, we propose to use *Siamese network*s[[20]](#footnote-21) 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 they 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. Using the same idea, we can identify all the nodes/links having the same color or texture coding.

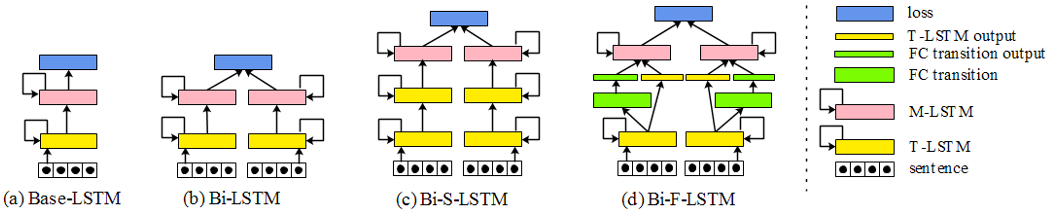


Figure : DL diagram with legend

# Code2Graph

## Overview

In the code2graph component, we have created methodologies and software artifacts to extract the Resource Description Framework (RDF) representations from the code included in Deep Learning (DL) publications. We focus on source code that uses Python as the main programming language and TensorFlow or Keras as the deep learning framework. Two main approaches have been completed during the Phase 1 of the project. More specifically, in the Computational Graph-based approach we created a pipeline to extract and simplify the RDF graphs by executing the code. On the other hand, our Lightweight Approach extracts the graphs via static analysis of the code. The updates pipeline architecture of the two approaches and our general methodology is described in Figure 8.

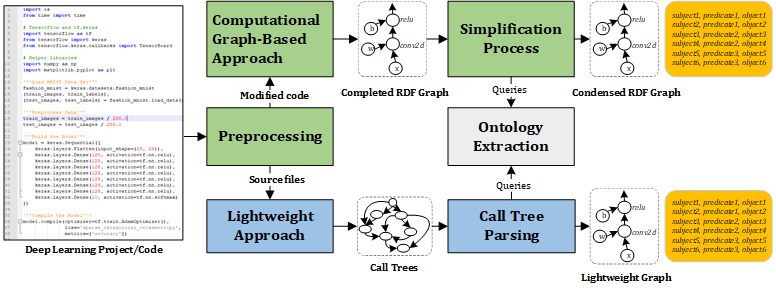


Figure 8: The pipeline of the code2graph demonstrating our formulations, models and implementation

Initially, we collect the source code that accompanies the DL publications obtained from the website: paperswithcode.com. Next, we use several preprocessing steps such as resolving requirements, injecting code into the main portion of the source code, etc. During Pase 1 (Milestones 1 and 2), we learned that fully automating the preprocessing in the **Computational Graph-based Approach** (shown in green colored boxes) is quite challenging process, since the developers may follow different programming styles and assumptions in their Python code. Therefore, manual effort is still needed in **Computational Graph-based Approach**, but it can generate a computation graph without the clutter of unrelated Python code. At the end of Phase 2 (Milestone 3), we found that the preprocessing can be fully automated if we adopt the **Lightweight Approach** (shown in blue colored boxes). However, the extracted RDF graphs will be mixed with information that may be irrelevant to the computation graph.

In the remaining of this section we will summarize each approach in mode details. We also include more details for the various components shown in Figure 5.

**Computational Graph-based Approach:**

We used TensorFlow’s built-in function FileWriter to extract the summary of computation from the TensorFlow-implemented code. The summary files are then converted into GraphDef objects with EventAccumulator, which is a built-in module in TensorBoard. The GraphDef object (a serialized version of the protocol buffer) is then turned into JSON, which contains enough information of DL architectures. The JSON-to-RDF Parser then generates the Complete RDF Graph using all the information such as code hierarchy, operation types, input-output relations, and names on the JSON graph.

***Simplification Process:*** The complete RDF graphs require a simplification process as they are too complicated to be useful in applications. The initial idea is to apply a Rule-Based Breadth-First Search (BFS) algorithm to the graphs. Whenever a node is reached, the algorithm will decide whether it should collapse or trim all the subsequent nodes. The algorithm can generate a simplified RDF graph with the preset of “unnecessary” and “interesting” node lists[[21]](#footnote-22), which are based on our general Machine Learning knowledge. However, this approach needs to be improved as it requires us to use a different set of keywords across different DL architectures. In fact, in phase II, we are working on an improved algorithm that will make the decision of simplifying the subsequent nodes based on the result of matching the node name with the keywords in the ontology.

***Ontology Extraction:*** As mentioned, we leveraged the well-defined structural definition from the TensorFlow Official website to describe the DL architectures. With the TFVocScraper, we then created a TensorFlow API module hierarchy and vocabulary as the ontology. OntologyManager is a class defined to store all the ontology information and enable the association of string keywords to their corresponding semantic meanings in the ontology. However, associating the node name with semantics is still a challenging task in code2graph and will be the one of the main focus of our effort during phase 2.

**Lightweight Approach:**

The main challenge for the computational graph-based approach is its scalability since it requires the code to be executed to an extent where we can stop the program and extract the summary. In Phase 2 we will be working towards automating the preprocessing process. Meanwhile, we have been exploring the possibilities of the lightweight approach in Milestone 3 and will be implementing in in large-scale during Phase 2. It is essentially a static AST-based analysis of code, which can allow us to acquire graphs without actually executing the code. The Python modules ast and pyan help us extract the ASTs from Python programs and generate the call graphs. Afterwards, we generate tree-like structures known as Call Trees that contain both sequential and hierarchical information.

***Call Tree Parsing:*** With the Call Trees, we then utilize TFCallVisitor, a class defined in code2graph to traverse the code trees and match the name of the function calls with the keywords defined in the ontology. TFC Visitor will append information to the node’s attribute if a TensorFlow call is matched with the name of the function call during the search. Finally, we generate the RDF triplets according to the attributes and the sequential information that each node carries in the resultant Call Trees.

## Updates and next steps

Our work in Phase 2 focuses on the following directions:

**Filtering architecture RDF from unnecessary data**

We are currently focusing on filtering extracted graphs so that only the knowledge graph relating to the scientific paper’s deep learning implementation is captured[[22]](#footnote-23). We will utilize these graphs to train our unsupervised learning algorithms to infer various parameters of the RDF graph.

**Generating code from RDF graph**

After the super-graph is created, the RDF graph for the code will be inferred. 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 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 shown in Figure 9.



Figure : Pipeline for converting the RDF graph to code

**Rule-based preprocessing of the RDF graph**

Our first step in converting the inferred RDF graph to code is 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. 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, when the super-graph is aligned and created, we will query the super graph to acquire additional possible attributes if available.

**Adding context/attribute to the RDF graph**

Once the RDF graph has been filtered, we add the context and attributes necessary for executing the computational graph corresponding to the RDF graph. For example, a 2D convolutional API has various arguments whose attribute must be defined (e.g., the size of the filters, output shape, data type of the output, etc., are some of the attributes corresponding to the conv2d operation).

**RDF graph to tf.Graph()**

Once the RDF graph has been pre-processed, 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 can be easy. 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 to submit the graph to the master TensorFlow core runtime. However, before submitting the graph to be tested 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.

**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. 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. 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.

**Knowledge graph embedding**

Our main focus in Phase 2 is to acquire large amount of knowledge RDF graph from the code repository to train an auto-encoder to embed the high-dimensional RDF knowledge graph to a smaller dimension embedding value. We are currently exploring 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, create 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 team. The auto-encoder architecture is shown in Figure 10.



Figure : KG embedding using auto-encoders

1. Other modalities that can also be considered are *equations*, and *tables*. [↑](#footnote-ref-2)
2. https://www.microsoft.com/en-us/research/project/microsoft-academic-graph/ [↑](#footnote-ref-3)
3. Although the dominant programming language is Python, developers also use C++, Java, and Julia frequently. Regarding DL frameworks it is our belief that the most popular is TensorFlow followed by Keras, PyTorch, Caffe, Theano, MXNET, CNTK. [↑](#footnote-ref-4)
4. e.g., a recurrent neural network is used for forecasting time series. [↑](#footnote-ref-5)
5. http://rat.nlplab.org [↑](#footnote-ref-6)
6. https://cso.kmi.open.ac.uk/home [↑](#footnote-ref-7)
7. https://spacy.io [↑](#footnote-ref-8)
8. Currently we have annotated only abstracts of the selected papers. [↑](#footnote-ref-9)
9. <https://cso.kmi.open.ac.uk/home> [↑](#footnote-ref-10)
10. https://hazyresearch.github.io/snorkel/ [↑](#footnote-ref-11)
11. [https://arxiv.org/pdf/1810.05102.pdf](https://eur01.safelinks.protection.outlook.com/?url=https%3A%2F%2Farxiv.org%2Fpdf%2F1810.05102.pdf&data=02%7C01%7Cdmitriy.fradkin%40siemens.com%7C3645b46d7c0346426b3b08d6d310d71e%7C38ae3bcd95794fd4addab42e1495d55a%7C1%7C0%7C636928466610010280&sdata=IGiucVQmAnUYsV7qSsnlbi0DrFovkAYip8ks86bUt7U%3D&reserved=0)  [↑](#footnote-ref-12)
12. Clark C., and Divvala S., “PDFFigures 2.0: Mining figures from research papers”, In Proc. Of the 16th Joint Conference on Digital Libraries, pp. 143-152 (2016) [↑](#footnote-ref-13)
13. All remaining diagrams in the paper are ignored. [↑](#footnote-ref-14)
14. https://arxiv.org/pdf/1506.01497.pdf [↑](#footnote-ref-15)
15. Poco, Jorge and Jeffrey Heer. “*Reverse-Engineering Visualizations: Recovering Visual Encodings from Chart Images*.” Comput. Graph. Forum 36 (2017): 353-363. [↑](#footnote-ref-16)
16. Vokhidov, Husan, Hyung Hong, Jin Kang, Toan Hoang, and Kang Park. "Recognition of damaged arrow-road markings by visible light camera sensor based on convolutional neural network." *Sensors* 16, no. 12 (2016): 2160 [↑](#footnote-ref-17)
17. P. Foucher, Y. Sebsadji, J.-P. Tarel, P. Charbonnier, and P. Nicolle, “Detection and recognition of urban road markings using images,” in Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on, oct. 2011, pp. 1747 –1752. [↑](#footnote-ref-18)
18. Liu, Wei, Jin Lv, Bing Yu, Weidong Shang, and Huai Yuan. "Multi-type road marking recognition using adaboost detection and extreme learning machine classification." In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 41-46. IEEE, 2015 [↑](#footnote-ref-19)
19. This is frequently the case in the vast majority of DL papers. [↑](#footnote-ref-20)
20. https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf [↑](#footnote-ref-21)
21. The term “unnecessary” (not interested) and “interested” are used in the naïve rule-based Breadth First Search (BFS) collapsing algorithms to simplify the complete computation graph generated directly from TensorFlow. The graph is a mixture of hierarchical, input-output, and attribute information of a DL model. Therefore, we envisioned that it may be infeasible to perform inference tasks using these graphs in the project’s later milestones. In our algorithm, we have used BFS search and have walked from the “root” of the graph and tried to check in the halfway if it is possible to prune or collapse the subsequent node entities. Whenever the BFS reaches a node in the interested word set, it stops and collapses all the subsequent node entities. On the other hand, whenever it finds an unnecessary word, then the subsequent nodes and itself will be pruned since it will not even be necessary to represent DL architectures. [↑](#footnote-ref-22)
22. All metadata that is specific to the C++ and Python programming languages are ignored. [↑](#footnote-ref-23)