**DARPA ASKE DCC – Milestone 3, 2019**

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

**Contents**

[1. Introduction 5](#_Toc4998809)

[2. Architecture Recap 5](#_Toc4998810)

[3. Datasets (needs update) 6](#_Toc4998811)

[4. Representation 6](#_Toc4998812)

[5. Text2Graph 6](#_Toc4998813)

[5.1 Annotations 7](#_Toc4998814)

[5.2 Named Entity Recognition 8](#_Toc4998815)

[5.3 Relation extraction 9](#_Toc4998816)

[5.4 Knowledge graph creation 9](#_Toc4998817)

[6. Image2Graph 9](#_Toc4998818)

[6.1 Extract figures from research paper: 10](#_Toc4998819)

[6.2 Figure classification: 10](#_Toc4998820)

[6.3 DL architecture diagram analysis: 12](#_Toc4998821)

[*6.3.1* Text Detection: 13](#_Toc4998822)

[*6.3.2* Arrow Detection: 14](#_Toc4998823)

[*6.3.3* Structural Analysis: 15](#_Toc4998824)

[6.4 Graph construction: 16](#_Toc4998825)

[6.5 Future Work: 17](#_Toc4998826)

[7. Code2Graph 18](#_Toc4998827)

[7.1 Proposed Methodology 20](#_Toc4998828)

[*7.1.1* Preprocessing 20](#_Toc4998829)

[7.2 Computational Graph-based Approach 23](#_Toc4998830)

[*7.2.1* Dataflow extraction: 23](#_Toc4998831)

[*7.2.2* Protocol buffer parser 24](#_Toc4998832)

[*7.2.3* Converting Json to RDF graph: 24](#_Toc4998833)

[7.3 Ontology and vocabulary extraction 28](#_Toc4998834)

[7.4 Light-Weight Method 29](#_Toc4998835)

[*7.4.1* AST Extraction 30](#_Toc4998836)

[*7.4.2* Call Graph Extraction 30](#_Toc4998837)

[*7.4.3* Graph Parsing 31](#_Toc4998838)

[*7.4.4* Sequence Extraction 32](#_Toc4998839)

[7.5 Computational Results 32](#_Toc4998840)

[*7.5.1* Computational graph extraction 32](#_Toc4998841)

[Test on simple graph 32](#_Toc4998842)

[7.6 Light weight graph extraction 35](#_Toc4998843)

[*7.6.1* Test on simple graph 35](#_Toc4998844)

[*7.6.2* Test on complicated graph 35](#_Toc4998845)

[7.7 Future Work 37](#_Toc4998846)

[*7.7.1* Filtering architecture RDF from unnecessary data 37](#_Toc4998847)

[*7.7.2* Generating code from RDF graph 37](#_Toc4998848)

[*7.7.3* Rule-based preprocessing of the RDF graph: 38](#_Toc4998849)

[*7.7.4* Adding context/attribute to the RDF graph: 38](#_Toc4998850)

[*7.7.5* RDF graph to tf.Graph(): 39](#_Toc4998851)

[*7.7.6* Evaluation of the Graph2Code: 39](#_Toc4998852)

[*7.7.7* Converting tf.Graph() structure to code: 39](#_Toc4998853)

[*7.7.8* Knowledge graph-embedding 40](#_Toc4998854)

[*7.7.9* Similarity measure and novelty measure in embedding space 40](#_Toc4998855)

[7.8 Discussion 41](#_Toc4998856)

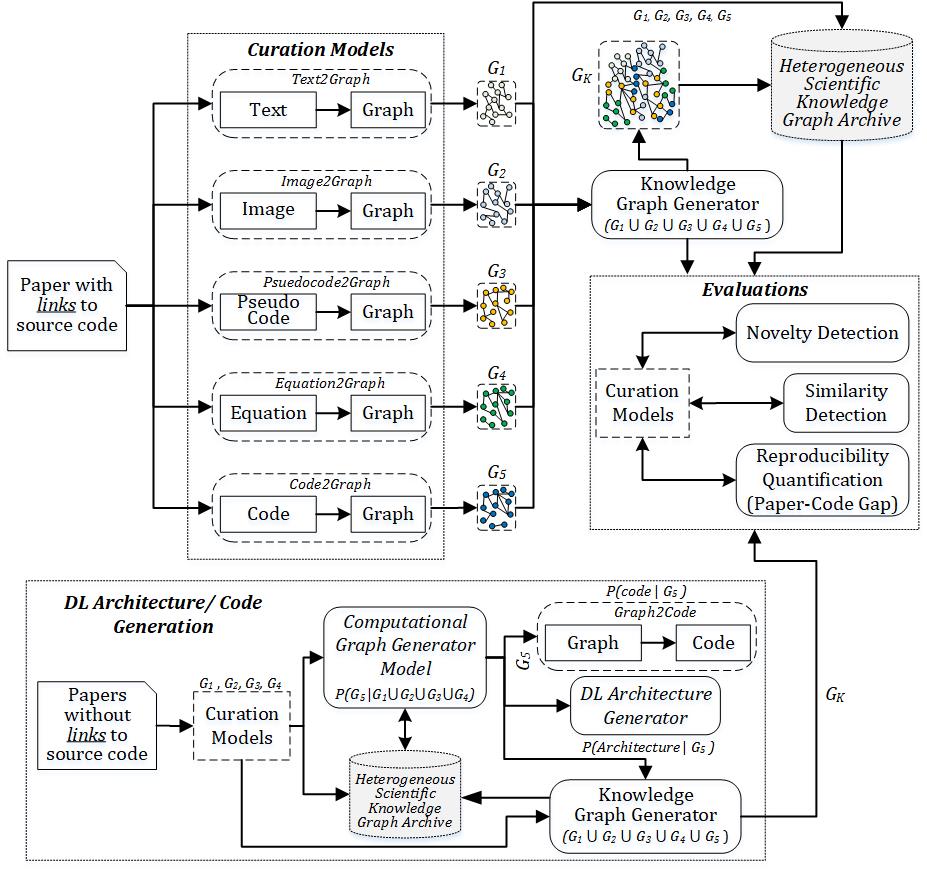
# Introduction

The purpose of this report is to provide a summary of the work performed under Defense Advanced Research Projects Agency (DARPA) project titled “Deep Code Curator (DCC)” under the agreement contract #HR00111990010 for the period from December 1st, 2018 to January 31th, 2019. The report provides details of the proposed prototype (Deep Code Curator), documents its basic components, and describes the basic learning algorithms used in various components.

In our quest to create automated deep learning models from pieces of the DL scientific process, we highlight the key contributions of our work

* **Text2graph**: We have built an initial state-of-the-art baseline architecture to extract textual knowledge graphs out of scientific publications. The process consists of extracting text from deep learning publications, annotate them using the web-based Brat tool, train deep learning models for NER and Relation Extraction, and finally produce the graph connecting entities with relationships. We have generated an implementable model that can generate graphs for paperswithcode [1] and show that it is extensible to any other paper in the arXiv repository.
* **Image2graph**: Given an image, we can characterize whether this image represents a deep learning framework architecture. After this characterization, we can extract specific components in the architecture and show how it is sequentially arranged with the other components. We also generate a knowledge graph to store this entire image-based information. Thus, we are able to synthesize the information from a variety of deep learning architectures (e.g., RNNs, CNNs, GANs, etc.).
* **Code2graph**: Synthesizing meaningful knowledge graphs that describe a scientific process from code is a challenging task. At this stage of the deliverable, we have been able to utilize syntactic structures of the python language (ASTs) and combine it with framework-based information (TensorFlow) to generate knowledge graphs that capture the high level architectural flow of a deep learning implementation in practice. We then apply this to all the papers from our reference dataset and have generated code-based knowledge graphs.
* **GitHub repo**: All the code developed during Phase 1 has been uploaded on GitHub at <https://github.com/deepcurator/DCC>. For each of the above modules we have added a Jupyter notebook that demonstrates its functionality. Instructions how to run the code are also included for each module. The repo also contains past milestone reports and links to the datasets.

# Architecture Recap



Our work proposes to extract the multi-modal components of a scientific paper along with the implementation specifics. We start out by creating knowledge graphs out of papers published in the deep learning community. The goal is to have an end-to-end representation of scientific ideas to implementation details, which in turn would lead to the creation of deep learning enabled deep model creation and suggestion. As stated in our previous report (for Milestone 1), we embarked on the creation of a scientific knowledge graph for each scientific document. This scientific knowledge graph would incorporate all elements of a scientific publication (text, images and code). To create such a unified knowledge graph, we proposed the extraction of each of these components as a knowledge graph (text2graph, image2graph, code2graph) in parallel. These components then will be aligned to form the unified scientific knowledge graph. In the following sections we describe the datasets, data representation, progress, challenges and way forward for each of these components.

# Datasets

There were no changes to the datasets used for text2graph and code2graph work from the previous report. Image2graph work involved additional datasets described separately. We identified two sources – (paperswithcode.com [16], and PWC [15]) for our initial development. We found that paperswithcode.com had the most structured curation. Each paper in paperswithcode is tagged with the pdf of the paper, the github repository and the data repository.

# Text2Graph

In this section we discuss our end-to-end approach that extracts entities and relationships from the text within articles (in PDF format) that describe DL methods, architectures and applications. Our framework is also able to generate a Knowledge Graph that integrates the extracted entities and relationships from each article. To achieve our goal we have (1) annotated the DL papers described in Section 3, (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 1. The text from the papers is extracted and annotated (entities and relations) using the web-based tool Brat[[1]](#footnote-2). Next the annotated text is used as input to train statistical models for named entity recognition and relation extraction. These models can be used to predict entities and relations in new text that has not been annotated before. The KG is generated by triple statement as a {*subject, predicate, object*}, where the *subject* and *object* are extracted by NER and the *predicate* is specified by the relation extraction component. Thus, we can view the triple statement as a {*entity1, relationship, entity2*}. In the next subsections we will describe more details for each of the components.

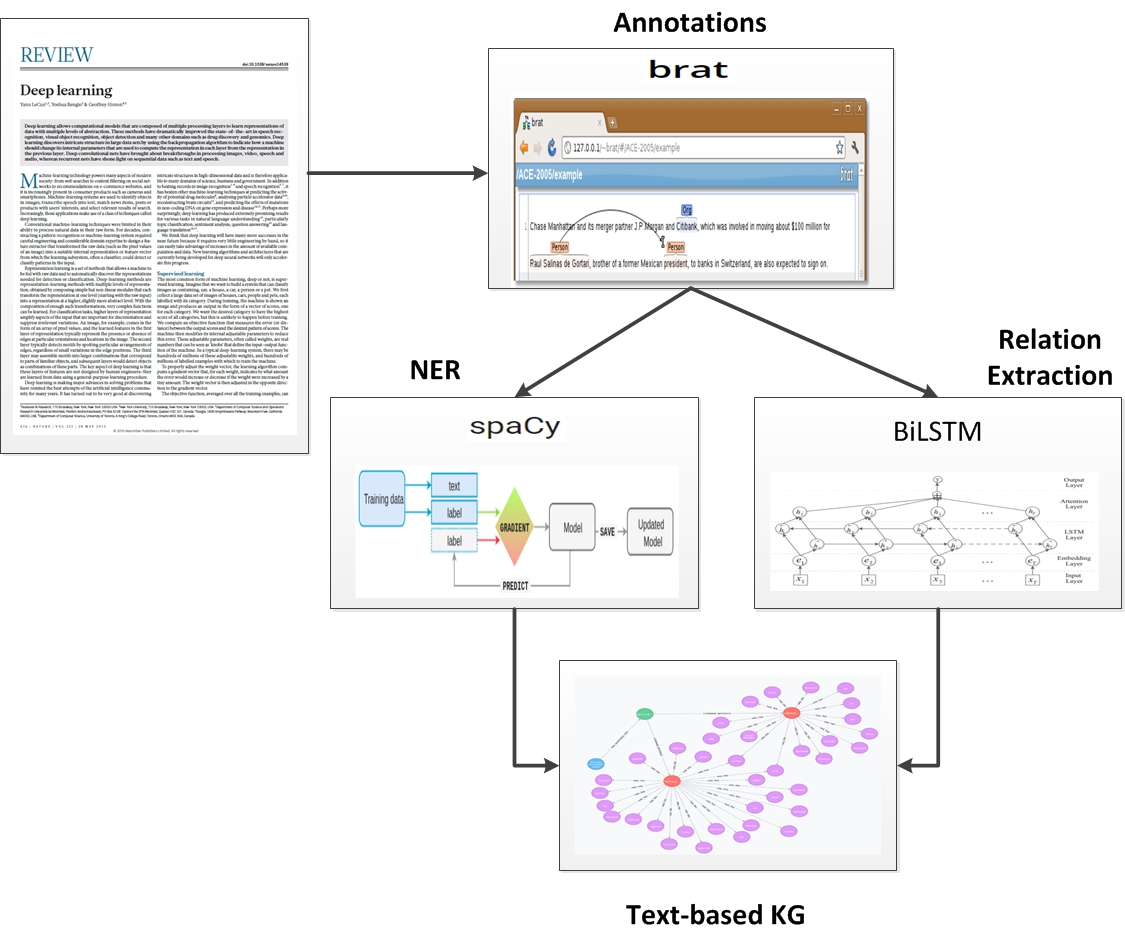


Figure 1 Text2Graph pipeline

## Annotations

We used the web-based tool Brat[[2]](#footnote-3) for annotating the collected Deep Learning articles. Brat is freely available, open-source and offers visualization features that enable the users to annotate faster. It supports a wide variety of configurable primitives such as: typed text spans (e.g., entity mention), binary relations, n-ary associations and free-form text notes, to name but a few. For the needs of our project we used Brat to annotate *entities* and *relations*. Brat creates annotations using the standoff format. For every document, it stores the text in a .txt file and its corresponding annotations in .ann file. The annotations are connected to specific spans of the text by using character offsets.

For example, in the sentence: *“we present foundations for using model predictive control (mpc) as a differentiable policy class for reinforcement learning in continuous state and action spaces”* the words “*model predictive control*” will be specified as an entity and will be identified by the offset ranges 33-57. Similarly, the term “*reinforcement learning*” will be specified as an entity and identified by the offsets 101-123. In the context of scientific articles related to deep learning, “*model predictive control*” can be identified as a method and “*reinforcement learning*” as a task. On the other hand, we can also specify the relation “Used-for” between those two entities as the *method* of “model predictive control” can be used in the *task* of “reinforcement learning”.

All annotations have the same structure, where each line contains a specific identifier, followed by an annotation type (e.g., an entity or a relation), followed by the start and the end of the annotation, and finally the text corresponding to the annotation type. For the example mentioned above the annotation file (.ann) will look as follows

*T1 Method 33 57 model predictive control*

*T2 Task 101 123 reinforcement learning*

*R1 Used-for Arg1:T1 Arg2:T2*

## Named Entity Recognition

Named Entity Recognition (NER) is the process that seeks to locate and classify named entities in text into a number of classes. Common classes are Person, Organization, Locations, etc. however, in the context of information extraction from scientific publications we needed to define our own entities. Following [19], we defined six entity types that are relevant for information extraction from Deep Learning papers:

* **Task**: ex. Information Extraction, machine learning, etc.
* **Method**: ex. Neural Network, Attention, CRF etc.
* **Evaluation Metric**: F1, Precision, Recall, ROC curve
* **Material**: ex. data, datasets, corpus, etc.
* **Other Scientific terms**: ex. dbpedia, Wikipedia, CoNLL, etc.
* **Generic**: ex. model, approach, algorithm, etc.

We used Brat to define these entities and annotate the DL papers in our dataset. To perform Named Entity Recognition we used the spaCy[[3]](#footnote-4) library. SpaCy has three main advantages: (1) it is considered one of the fastest libraries for large-scale information extraction, (2) it provides very efficient machine learning algorithms for various NLP tasks, and (3) 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 within the learning model.

To train our model spaCy needs sentences in the format *(text\_list, annotations\_list)*, where *text\_list* contains all the text extracted from the DL papers in a sentence format, and *annotations\_list* is the list of the annotations corresponding to each sentence. To populate that structure we use the information obtained from the annotated files produced from Brat. More specifically, in order to use the example used in the previous section as input for training of our model will have convert it to the following format:

(“*we present foundations for using model predictive control (mpc) as a differentiable policy class for reinforcement learning in continuous state and action spaces”*, {“entities”: [ (33, 57, *Method*), (101, 123, *Task*) ] } ).

To make the most of the sentences comprising our training examples, we used several techniques that can increase the performance of our NER model. For example, at each iteration of the learning algorithm we reshuffle the training data in order to ensure the model does not make any generalizations based on the sequence of the training examples. We also explored adjusting the dropout rate which helps improve the learned model. The main effect of the dropout rate is to randomly ignore specific features of the input sentences, in order to make it harder for the model to memorize the training dataset 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[[4]](#footnote-5).

Once the NER model is trained it can use it to annotate new sentences taken from papers describing DL methods. The output consists of a list of the predicted entities together with their text spans.

## Relation extraction

To generate triples in the form of {*entity1, relationship, entity2*}, we first need to extract formal relationships from scientific papers. This involves the task of relationship classification. While large scale knowledge graphs in general don’t classify relationships, we look at classifying relationships because that gives as a basis to align graphs in the next phase. In our view textual knowledge serves as a basis for expressing provenance and facts that augment both the images and code. Thus, we believe the texts serve as a reference point for understanding the scientific models and the results. As an initial architecture exploration, we decided to annotate the papers from paperswithcode [1] with the following relations gleaned from the Semeval-10 task on knowledge graph creation from scientific text. The following relationships were identified as being part of the scientific text:

* **Used-for:** B is used for A, B models A, etc.
* **Feature-of**: B belongs to A, B is a feature of A, etc.
* **Hyponym**-of : ex. Text Extraction is a NLP problem
* **Part**-of: ex. Our system includes two models – A and B
* **Compare** : Comparing two works
* **Conjunction** : Symmetric relation
* **isA** : ex. DNN is a type of artificial neural network
* **sameAs** : ex. NMT, otherwise known as neural machine translation.

While six relationships came from [19], we added two more relationships, **isA** and **sameAs**, which were used in earlier versions of Semeval[[5]](#footnote-6). These two relationships enable easy merging with large scale knowledge graphs to enhance further fine-grained hierarchical querying. The annotation process was carried out in the same tool. Our annotation process thus followed a multi-task setup that involved identification of entities and relationships. We annotated the abstracts from 100 papers in paperswithcode with the following relationships. A total of 1274 sentences were annotated and these were found to be very effective in training and end to end system. Since our setup is a multi-task setup, we also negated the issue of cascading errors between tasks.

**Architecture Components:**

Following state-of-the art work on using deep learning for natural language processing, we proceeded to use the same way. Part of the reason for this was the fact that one could reduce the number of hand crafted features that were part of Natural language processing tasks. Since we wanted a quick knowledge graph that captured the essence of the scientific text we utilized a Bidirectional LSTM along with a neural attention mechanism to capture the relationships in the graphs[19]. Without any lexical features for just about 100 papers, this model captures the essence of the scientific information that we intended to capture. The components of the model can be described as follows:

1. Input layer: The input sentence to the model.
2. Embedding layer: Each word in the sentence is mapped to a low dimension vector.
3. LSTM layer: BiLSTMs are in general used for information extraction tasks in NLP. The main idea is to use an adaptive gating mechanism to decide the degree to which the LSTM units can keep the previous state and memorize the extracted features of the current data. For our work we use a LSTM variant that introduces weighted peephole connections from the Constant Error Carousel to the gates of the same memory block.[20]
4. Attention Layer: Attention mechanism for improving the focus on words that have decisive effect on the classification of relationship. This also helps in capturing the most important semantic information in a sentence.

We implement a version of [19] provided with a jupyter notebook to test relationship sentences. On semeval data our accuracy matches the F-1 accuracy of 82%. However, on our data the accuracy drops to 74%. We put this down to the lack of training data, which can improve the accuracy of extraction.

## Knowledge graph generation

Knowledge graph generation from the relationship extraction is a step that involves construction of the knowledge graph by building triples that take in as arguments entities in the sentence and the relationship that the sentence was part of. For example, in the following sentence, "*finally, we can use recurjac to evaluate the robustness of neural networks, by giving a certified lower bound within which no adversarial examples can be found*”, the generated knowledge graph triple would look like:

(Recurjac, used-for, robustness)

# Image2Graph

In this work, we present a novel end-to-end framework that automatically localizes all figures from a research paper, classifies them, extracts the content of the DL architecture figures and represent it in the form of a graph. Given the PDF of a deep learning research paper, image2graph module consists of four major steps, as shown in Figure 2: (i) extract all the figures from a research paper, (ii) identify figures showing DL model diagram, (iii) perform diagram analysis, (iv) construct a graph representing the diagram. As part of milestone 2 work, we developed algorithms for performing tasks (i) and (ii). Algorithm for performing task (iii) is under development. Here we describe the baseline approach with some initial results. Apart from the algorithms, we also created a labeled dataset to train the classifiers and test their efficacy.

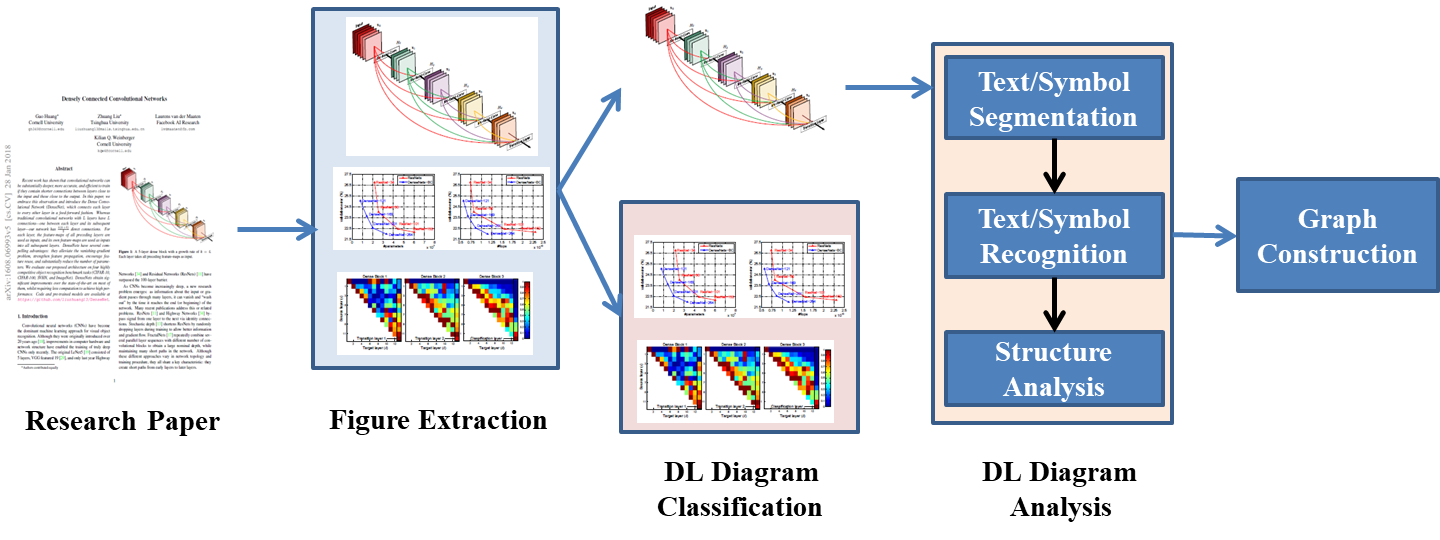
****

Figure 2The architecture of the image2graph module

## Extract figures from research paper:

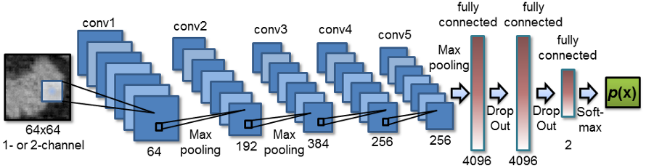
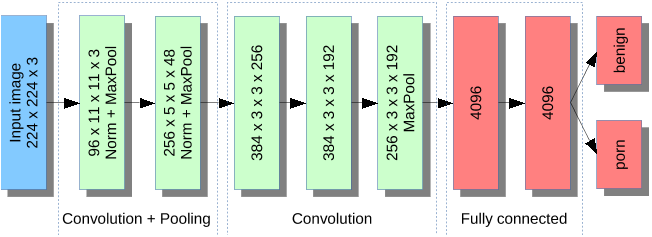
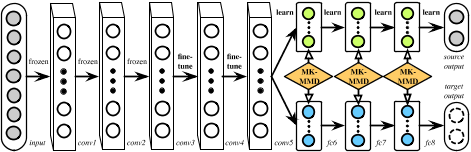
We used PDFFigures 2.04 [6] tool for automatically extracting a list of figures from a research paper. We downloaded 1000 papers from arXiv.org using “deep learning” as the input query. 8310 figures were extracted from these 1000 downloaded papers.

## Figure classification:

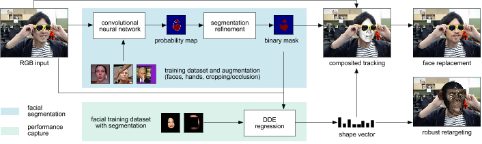
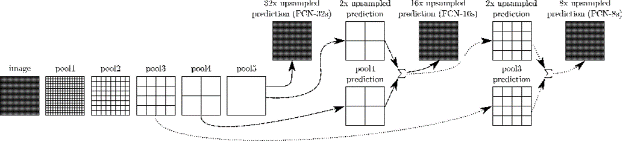
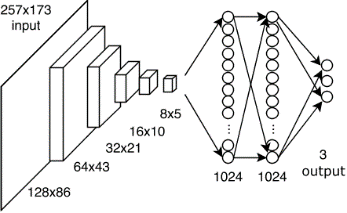
Figure classification is performed in two steps. First, only the DL diagrams are identified and retrieved from all the figures and tables extracted from a research paper. We develop a binary classifier to perform this task.

It has been observed that the DL architecture diagrams typically do not follow any definition and show extreme variations. We categorized the model diagrams into 6 broad types as follows: (i) Neurons plot: the classical representation of a neural network with each layer having circular nodes inside them, (ii) 2D Box: each hidden layer is represented as a 2D rectangular box, (iii) Stacked2D Box: each layer is represented as a stack of 2D rectangular boxes, describing the depth of the layer, (iv) 3D Box: each hidden layer is represented as a 3D cuboid structure, (v) Matrix Box: each hidden layer is represented as a matrix, and (vi) Pipeline plot: along with the DL model design, the entire pipeline and mostly some intermediate results of image/ text is shown as well (see Figure 3).

Next, we train a fine-grained six-class classifier to classify the figure into one of the six broad categories.

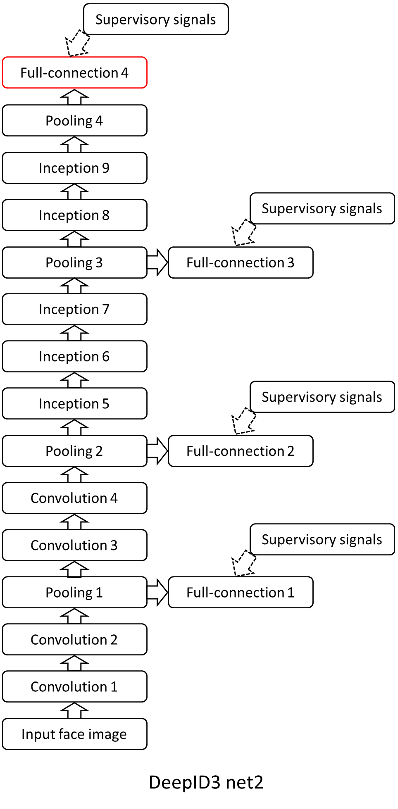
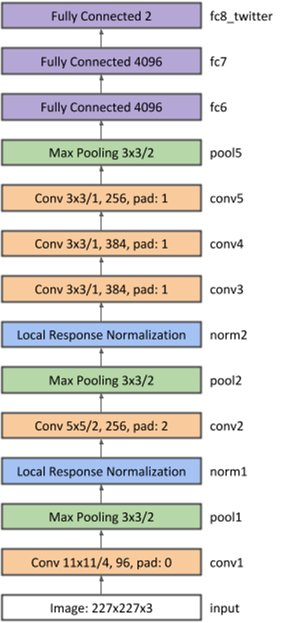


1. (b) (c)



(d) (e) (f)

Figure 3: Categorization of the DL design diagrams into six classes: (a) Neurons plot, (b) 2D box, (c) Stacked2D box, (d) 3D box, (e) Matrix box, (f) Pipeline plot.

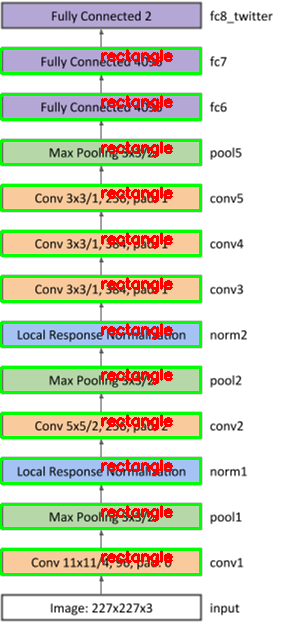
(a) (b)

Figure 4: DL 2D diagram (a) line diagram, (b) blob diagram

## DL architecture diagram analysis:

Once the figure to be analyzed further is determined, we apply component extraction algorithm to identify the building blocks present in a DL architecture diagram. In this section, we explain the details of the approach for a 2D Box type, as shown in Figure 4, while similar approach could be extended to other types, as well. DL architecture analysis involves identifying the nodes first, followed by the edges connecting the nodes. This module is still under development and we are experimenting with several possible approaches. Here we report traditional image processing-based approach to detect the nodes.

**Node Detection:** We first apply image preprocessing technique to enhance the image quality by color balancing, contrast and brightness adjustment. Then thresholding is applied to binarize the image. Application of contour detection technique on the binary image of 2D line diagram image is shown in Figure 4 (a).

1. (b)

Figure 5: Node/Layer detection

However, as the diagram images often contain touching or overlapping objects, application of contour detection technique on the binary images will not extract each individual nodes from the image. So, we apply iterative region growing technique (which is especially useful when extracting touching or overlapping objects) to identify closed contours of the nodes as shown in Figure 5 (b).

The contours extracted from the two above mentioned techniques may have overlapping bounding boxes corresponding to the same node. So, we apply non-max suppression technique to filter out weak bounding boxes based on their solidity indices.

### Text Detection:

Once the nodes are extracted, the text in each node/ layer is obtained in two stages: text detection and text recognition. We apply EAST text detector [<https://arxiv.org/abs/1704.03155>] to extract the text regions in the whole image. Next, we perform OCR using Tesseract7 on the extracted text regions. However, DL architecture diagrams contains specific acronym which may not be available in general dictionary. In such cases, traditional Tesseract performs poorly. To improve OCR performance, we created a new DL dictionary with possible words/acronyms found in DL diagrams. Use of this DL-dictionary helps in spell correction and improve overall OCR performance. The results of text detection from four diagrams are shown in Figure 6.

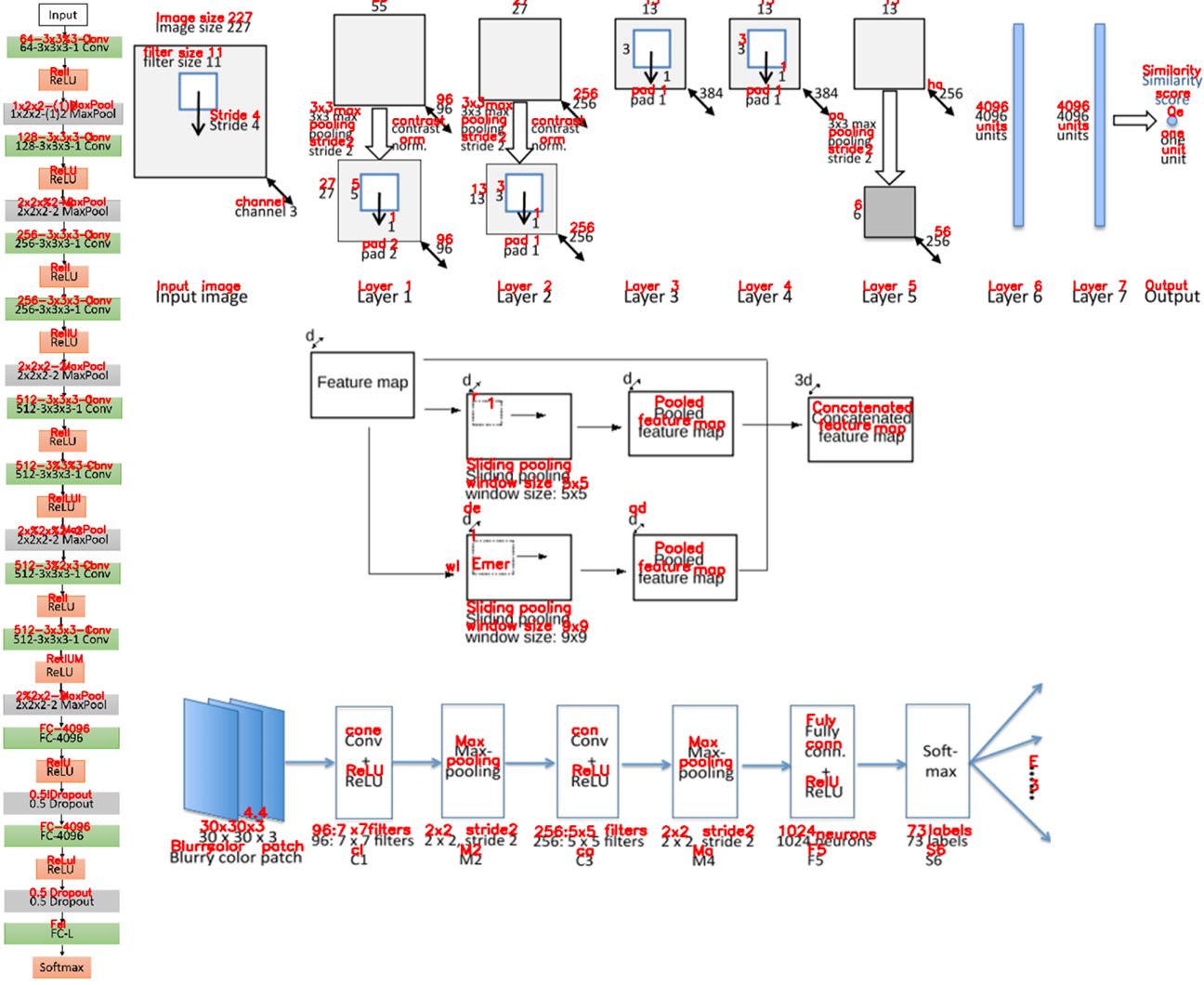
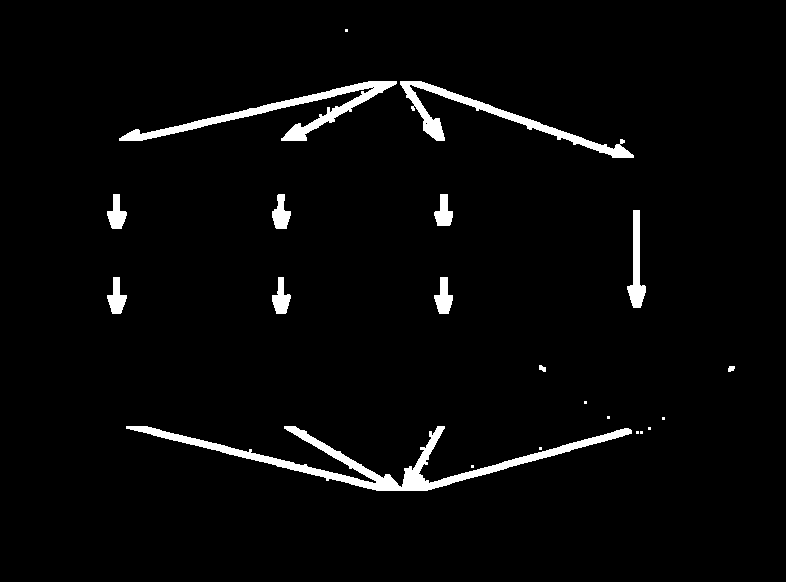
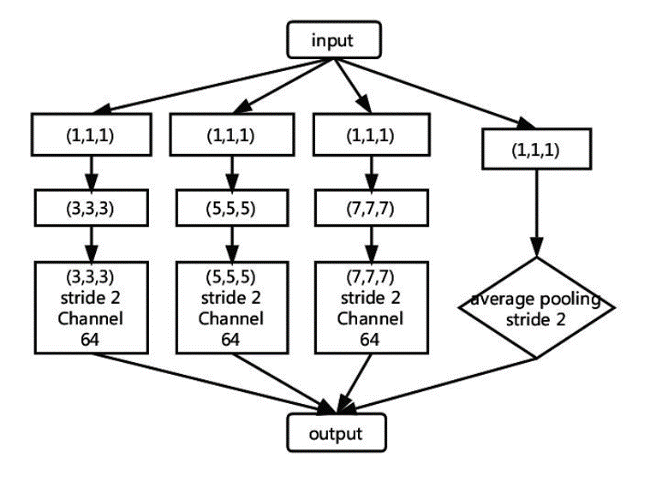


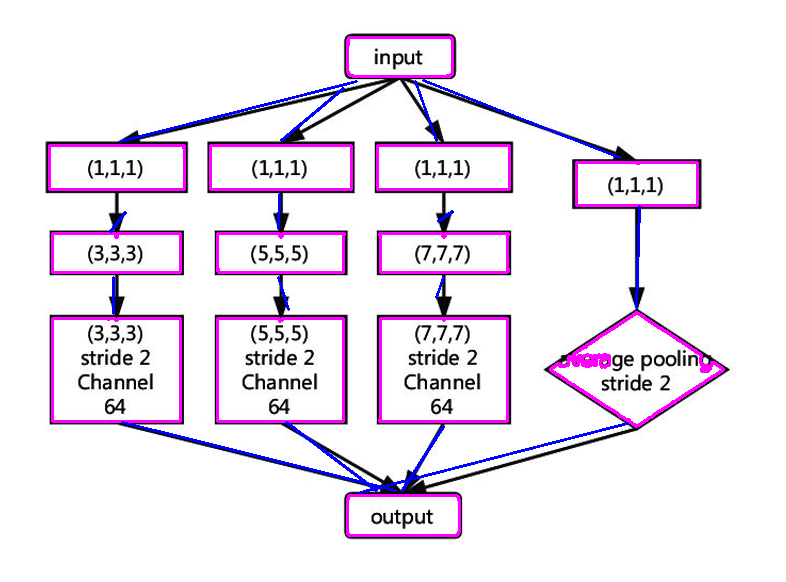
Figure 6: results of text recognition

### Arrow Detection:

Arrows in architecture diagram give important information about the flow of the entire DL model design. To detect the arrows, all the detected nodes and the text regions are masked out from the figure (see figure 6(b)). Next, Hough line detection algorithm is applied to detect the arrow lines, as shown in figure 6(c). The direction of the arrow is obtained by analyzing the pixel distribution of the contour corresponding to a detected arrow line. As shown in Figure 7(c), the proposed approach could also handle branchings and forking in a design flow diagram.



1. **(b)**



**(c)**

Figure 7: (a) Original Image (b) Segmented image with arrows (c) Arrow connections among detected nodes

### Structural Analysis:

Structural analysis is used to find spatial and logical relations among the node and text candidates and to generate the final interpretation. General flowchart has its own structure features: (a) it can be drawn in different orders; (b) each two non-arrow symbols are connected with an arrow; (c) strong precedence relationship exists between each two symbols; (d) the arrows are directive. While (a) (b) are called physical structure of a flowchart, (c) (d) are called logical structure of a flowchart. Existing flow diagram analysis works utilize this structure grammar to perform structural analysis. However, DL diagrams do not strictly follow these structures (e.g. each two non-arrow boxes may not be connected with an arrow). So, we employ a combination of logical structure of a DL diagram with graph grammar to perform the flow detection.

The node, text and arrow contours are first sorted based on the location and direction. According to [5], a layer description is generally provided within a detected block or in its vicinity. So, each text box is tagged with the nearest neighbor node (see Figure 8). However, if the nearest neighbor node is too far away (beyond a pre-defined threshold), the text is not tagged with any node and considered as an independent text component (yellow text boxes in the figure). In such cases, it has been observed that the text generally describes network name or provide common information about more than one node.

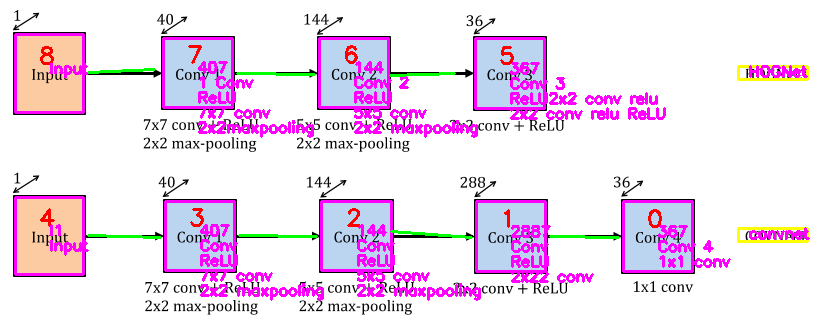


Figure 8: All the nodes extracted from an example DL diagram are shown here. The node numbers are colored in red. The text description associated with a node is depicted with pink color within the node. The text box in yellow color shows text description not tag

Next, the start and end points of the arrows are analyzed to detect connectivity between two nodes/layers. We create a grammar defining the list of possible next layers for a given current layer. The arrow directions along with possible valid next node/layer information is used to generate valid flow detection.

## Graph construction:

After detecting DL design flow, a computational graph is created with following relationships describing each node or pair of nodes: “is type”, “has description”, “connected to”, “followed by”, “has input”, has output”. Figure 9 shows one graph relationship constructed from Figure 8.



Figure 9: Graph relationships extracted for DL diagram in previous figure

## Future Work:

* In the next stage, we will develop node detection algorithm beyond 2D DL diagrams.
* One possible approach is to automatically detect the objects/symbols [17, 18,19] in an image and the relationship between these objects/symbols. Recently, deep learning based technique namely faster R-CNN has been applied for this purpose. We plan to 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 and providing 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 (GAIN) [11] framework, regularization on the attention maps will be used to guide the learning of detecting the DL building blocks more efficiently and effectively.
* Arrow detection algorithm will be extended to take care of curved arrows.
* The structural analysis of DL diagram is a challenging research problem as there is no predefined grammar of DL diagram. Some of the challenges that we aim to solve in the next phase of the program are described below:
  + Complex node connection with overlapping arrows (Figure 10(a))
  + Interpretation of common description about a set of layers (Figure 10(c))
  + Interpretation of color code in arrows (Figure 10(b)) or in nodes (Figure 10(d))
  + Flow detection in the absence of arrows in DL diagrams (Figure 10(d))
  + Decoding legend (see Figure 10(e)) even when it has not been specifically mentioned in the diagram figure

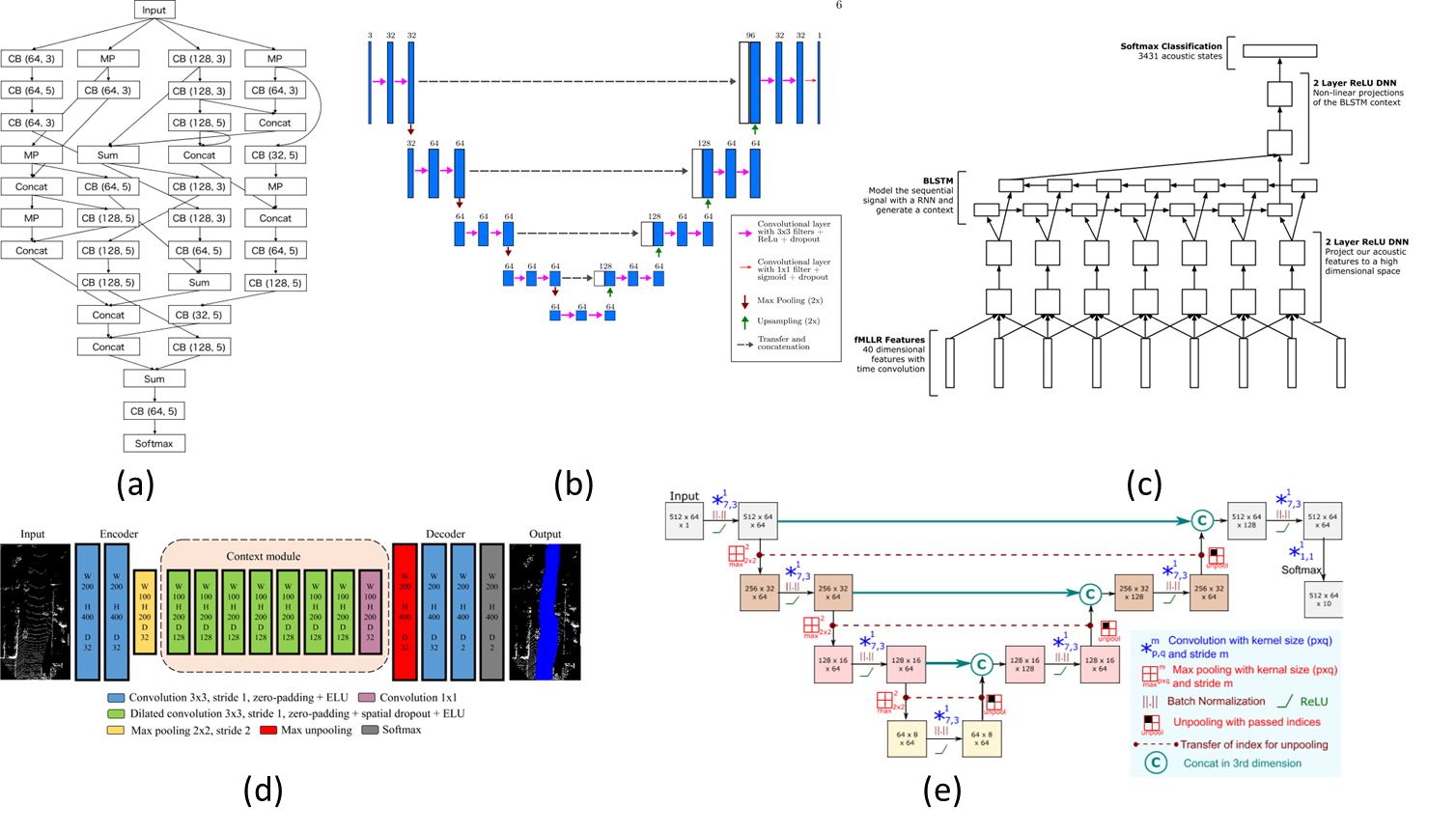


Figure 10: Challenging examples for DL diagram interpretation: (a) complex node connection (b) color-coded arrows with specific meaning, (c) text description over a set of layers, (d) color-coded nodes without arrow connection, (e) figure with complex legend and

# Code2Graph

The main purpose of this part of the code2graph component is to extract knowledge graphs 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 resource description framework graph (RDF) are extracted from the deep learning architectures written in python and utilizing frameworks such as TensorFlow and Keras. The main aim in extracting the knowledge graphs from the code is threefold. Firstly, it will serve the completion of the information extracted from the text and the images of the scientific graph. Secondly, this aggregated and fused graph will serve as the graph from which we aim to perform *novelty* and *similarity* detection among other scientific papers, and train stochastic models to aid in completing the inferred code graph, utilizing the traditional rule-based inference methodologies used in extracting information from the knowledge graphs. Thirdly, to create models that 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 [7], conventions [8], patterns [9], algorithm [10] etc. However, the works that are closest in converting the code data structure to meaningful graph or sequence structure for summary, documentation, etc., are [11, 12,13,14]. These methods use various approaches to convert the source code to graphs structures (e.g., abstract syntax tree (AST), data flow graphs, control flow graphs, paths in AST, augmented AST, contextual flow graphs using the concepts of program dependence graph (PDG)). However, all these approaches do not consider aligning the code with information extracted from texts or images. Hence, the graph structure generated is not scalable for effective super knowledge graph construction.

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 the graph2code process), while still being abstract enough to be aligned with the knowledge graph generated by text2graph and image2graph.
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.

The scientific papers that utilize the TensorFlow runtime which is a cross-platform library where all the computations are represented as a dataflow graph. It utilizes the concept of client and master, where client consists of a user level code (written in Python) and the master consists of the core runtime kernel implementations (written in C++). The scientific codes are 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.

One of the advantages of using TensorFlow runtime library is that it converts all the user’s program into the dataflow graph. That 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 us consider a matrix multiplication (y=x\*w) using TensorFlow:

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

To calculate this operation, TensorFlow generates the following dataflow graph

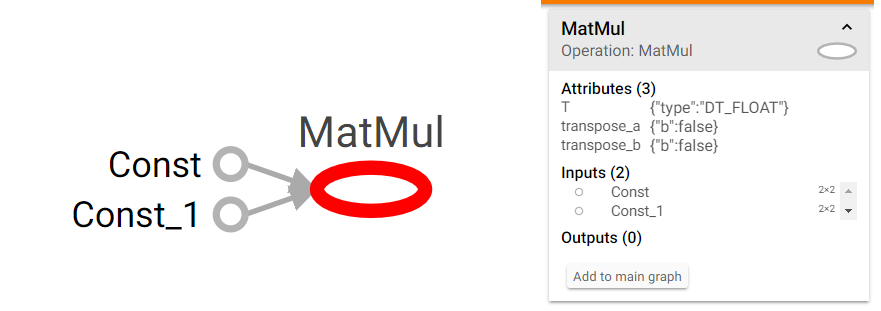


Figure 11: TensorFlow graph visualized using Tensorboard

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 a Resource Description Framework (RDF) graph.

## 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.



Figure 12: The code to graph extraction pipeline

The pipeline for the knowledge graph extraction is shown in Figure 12. Initially in milestones 1 and 2, we focused on extraction of knowledge graph through extraction of computational graph. However, we have also explored the possibility of knowledge graph from static call graphs, which we call the Abstract Syntax Tree (AST) approach (highlighted in blue color). 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, they are called abstract. AST data structure is normally used by compilers to represent the structure of the program. In the subsequent sections we will discuss the methodologies in both directions.

### Preprocessing

The paper-datasets that we consider (taken from [15] and [16]) 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. There are several steps carried out to perform the pre-processing so that both the computational and the light-weight AST-based method can be utilized to extract the knowledge graph. These steps are explained as follows:

#### Python version check

The code repositories contain codes that are written in Python 2 and 3. In order to check the version the python script “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. Then a virtual environment is created with the specific version of python to further extract the information from the code.

#### Resolving requirements

After acquiring the virtual environment, the necessary packages required for running the code repository are installed. Normally, these requirements are listed in the README.md file 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 help both 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 search in the python package index[[6]](#footnote-7) to get the installation details.

#### Code injection

For extracting the computational graph from the code, we need to partially compile up-to the point where all the necessary deep learning architecture components are added. In Keras, this is easy to find as after all the components are added a “compile” method is called to configure the model for training. However, for a purely TensorFlow implementation this is not the case. For TensorFlow, the sess.run() needs to be searched as the computational graph must be compiled before it can be called. In order to achieve the computational graph extraction, a small portion of code is injected to original code either after “compile” method for Keras, or before “sess.run()” for TensorFlow. We realize the above mentioned functionally as follows:

1. **class** CompileVisitor(ast.NodeVisitor):
2. **def** \_\_init\_\_(self):
3. self.found = False
5. **def** visit\_Call(self, node):
6. **if** isinstance(node.func, ast.Attribute) **and** node.func.attr == "compile":
7. self.found = True

*Code Snippet: the class definition of NodeVisitor that inspects whether an instruction calls “compile”.*

1. **def** inject():
2. inject\_code(CompileVisitor, code\_tree)
3. print(astor.to\_source(self.code\_ast\_tree))
4. **def** inject\_code(class\_visitor, node):
6. location\_to\_be\_inserted = []
8. **for** idx, obj **in** enumerate(node.body):
9. visitor = class\_visitor()
10. visitor.visit(obj)
11. **if** visitor.found:
12. **if** isinstance(obj, (ast.Expr, ast.Call)):
13. location\_to\_be\_inserted.append(idx)
14. **elif** isinstance(obj, (ast.If, ast.For)):
15. inject\_code(class\_visitor, obj)
16. **else**: # should not reach
17. **print**(astor.dump\_tree(obj))
19. **if** len(location\_to\_be\_inserted):
20. **for** idx **in** location\_to\_be\_inserted[::-1]:
21. node.body.insert(idx+1, self.inject\_code\_ast\_tree)

To extract the serialized version of the models in code repositories, we inject the script, which is shown below, and then extract the event summary file. In the script, we first get 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. This script is injected after the model is compiled and before the training starts in the code. If the code is using Keras, there will be an explicit compilation of the model. However, this is not the case for TensorFlow native. In this case, the model components are added to the session and the session is run without compiling the model explicitly. Thus, in Keras, we inject this script after model.compile() line, and in TensorFlow, we inject it before sess.run() line.

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()

#### 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. This makes 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 may 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 DL 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, and after that we can determine which model is the one used in the paper.

The second challenge is 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 settings in a file such as the “readme.md”. Scanning through the “readme.md” might give us some hints on the way to run the script. However, some developers might just not include this information. Hence, it still requires manual effort to find how to run the script.

To automate the full process in pre-processing 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 design the Lightweight graph extraction approach that can generate RDF graph by static code analysis. We will describe both approaches in the later sections.

## Computational Graph-based Approach

### Dataflow extraction:

To be able to store the deep learning architectural information from code, we first of all 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:

1. 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.

### Protocol buffer parser

In order to read the even file stored by the tf.summary.FileWriter(), we have traced the tensorboard code from the github repository and located the function that understands the event file. The python module which will allow us to do this is called event\_accumulator which can be imported from tensorboard . Next, we found out that we can extract the protocol buffer using the following script:

1. accumulator = event\_accumulator.EventAccumulator(logdir)
2. accumulator.Reload()
3. graphDef = accumulator.Graph()

Previously, we discussed that the 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(...)) .

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.

### Converting Json to RDF graph:

Since the JSON string we extracted by calling tf.GraphDef() has an immensely complex structure as mentioned earlier, we had to parse its hierarchical structure first to convert it to RDF [12]. We have utilized a python library for rdf representation and processing called rdflib. We have implemented a class structure that is responsible for this called Json2RDFParser. Then we have utilized the extracted ontology from the TensorFlow modules hierarchy to filter the complicated graph. Throughout the explanation, the following example of a json node (dense/bias/Adam/Assign) will be used for better depicting how the tasks of parsing are done.

1. {'attr': {'\_class': {'list': {'s':['...==']}},
2. 'dtype': {'type': 'DT\_FLOAT'}},
3. 'input': ['dense/bias/Adam', 'dense/bias/Adam/Initializer/zeros'],
4. 'name': 'dense/bias/Adam/Assign',
5. 'op': 'AssignVariableOp'}

The JSON format of the computational graph is a dictionary structure. When a deep learning architecture is compiled, the corresponding computational graph will have a set of nodes. Nodes basically are the fundamental building blocks of the Tensorflow graphs, whereby each node represents an operation. In order to convert the computational graph into RDF format, the parser iterates through each field inside the nodes. As the example above shows, the node, with name “dense/bias/Adam/Assign, consists of five fields: ‘attr’, ‘input’, ‘device’, ‘name’, and ‘op’. To parse this structure, we have written different procedures. A glimpse of the main parsing routine is shown below, followed by the step-by-step explanation for its working procedure.

1. **def** parse(self, jsonGraph):
2. ...
3. self.parse\_node\_hierarchy(sorted\_nodes)
4. self.parse\_node\_device(sorted\_nodes)
5. self.parse\_node\_op(sorted\_nodes)
6. self.parse\_node\_input(sorted\_nodes)
7. self.parse\_node\_attr(sorted\_nodes)
9. self.simplify\_RDF\_graph()

* Parse node hierarchy:

To create the RDF graph, we first constructed the hierarchy of the nodes. In this hierarchy there are nodes which are actually described in the computation graph, and the ones that do not have definition but are the parent node of multiple operations. We call these types 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 aforementioned 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>

* 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>

* Parse node component: “op”

Similar to 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>

* Parse node component: “attr”

Similar to 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:

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:

1. Format: <<node\_name>, <attribute\_name>, Literal<value>>
2. Example 1: <dense/bias/Adam/Assign, dtype, "DT\_FLOAT">
3. Example 2: <dense/bias/Adam/Assign, \_class, "...==">

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.

* 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 (BFS) 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.

1. # BFS to search nodes.
2. fringe = [self.root]
3. visited = []
4. **while** fringe:
5. node = fringe.pop(0)
6. visited.append(node)
7. **for** o **in** self.RDF\_graph[node:RDFS.member]:
8. **if** sum( [1 **if** w **in** str(o).lower() **else** 0 **for** w **in** unnecessary\_words]):
9. **pass** # stop this branch of BFS is the word is interested.
10. **elif** sum( [1 **if** w **in** str(o).lower() **else** 0 **for** w **in** interested\_words]):
11. visited.append(o)
12. **else**:
13. fringe.append(o)
15. words\_bank = visited

After the BFS, words\_bank contains the node 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 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.

## 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 TensforFlow 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 13). It can be noticed in Figure 13 what types of nodes to focus on extracting can be listed by populating the custom vocabulary with leaf nodes (which are actually 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. It can be also be visualized using python plotting libraries such as networkx as below.

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 namescopes between queried string and terms stored in ontology\_manager and generate a ranked list of the most related terms in the ontology.

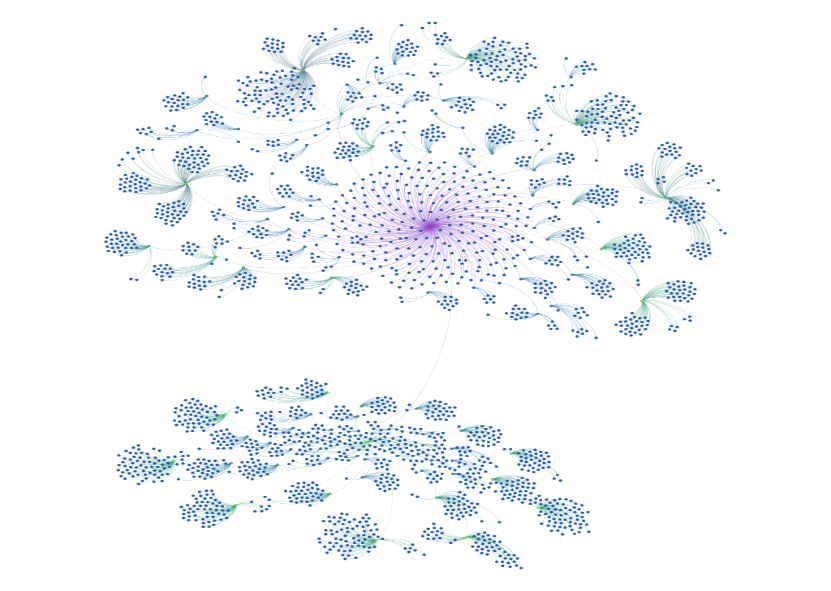
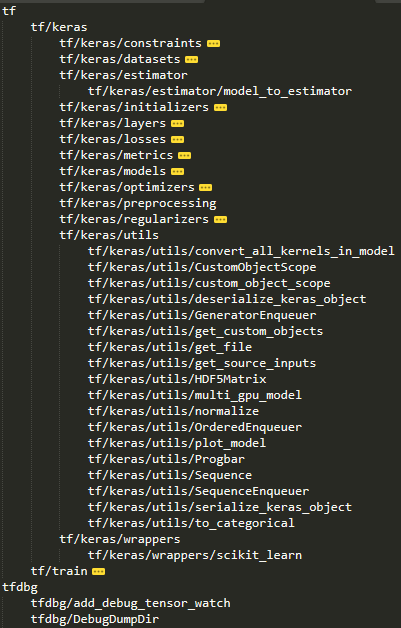


Figure 13: Ontology structure extracted from the TensorFlow repository. It can be noticed that everything stems from tf, and all the instance are arranged in Tree (for example Dense, Conv1D, Dropout are children of Layer node, where “layer” is a virtual node).

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'})

## 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 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 invoke 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 (1) generating the call graph, (2) generating the call tree, (3) lightweight RDF graph generation and (4) generating the TensorFlow word sequence.

### AST Extraction

As mentioned earlier, Abstract Syntax Tree (AST) is a data structure which captures the syntactic structure of the source code for the specific programming language. AST data structure is normally used by compilers to represent the structure of the program. AST trees are extracted using the **ast** [17] 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.

### 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 [18] 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 the 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 14 Sample call graph

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.

### 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.

### 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.

## Computational Results

### Computational graph extraction

### Test on simple graph

The raw computational graph extracted from the MNIST example is shown as follows:

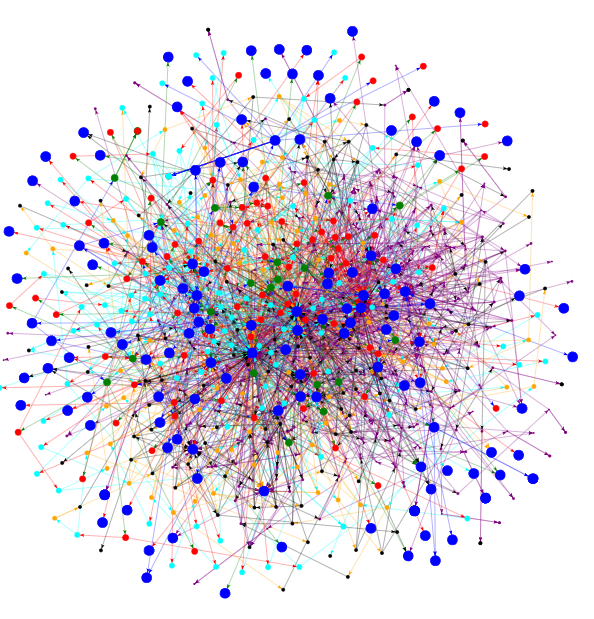


Figure 15: Raw computational graph extracted from MNIST example.

As mentioned earlier, with all the attributes RDF graph is too complicated to be used in applications (see Figure 15). There are usually thousands of triples in each RDF (which varies based on different deep learning models). The simplified graph is shown in Figure 16. In addition In Figure 17 we show the simplified graphs of two more complex DL architectures, a CNN model on the left and a Generative Adversarial Network (GAN) on the right.

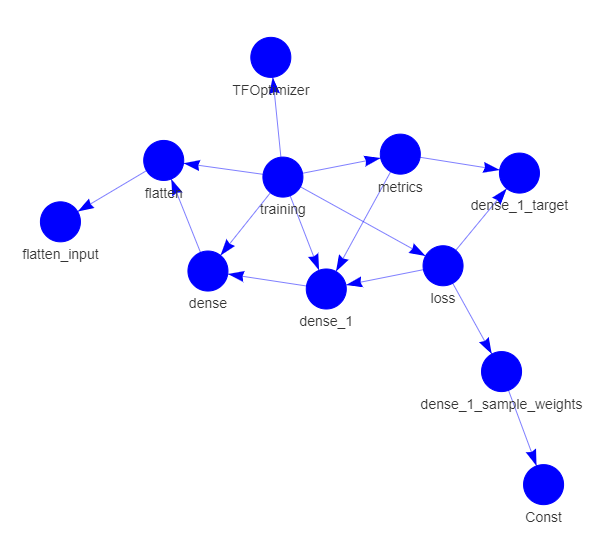
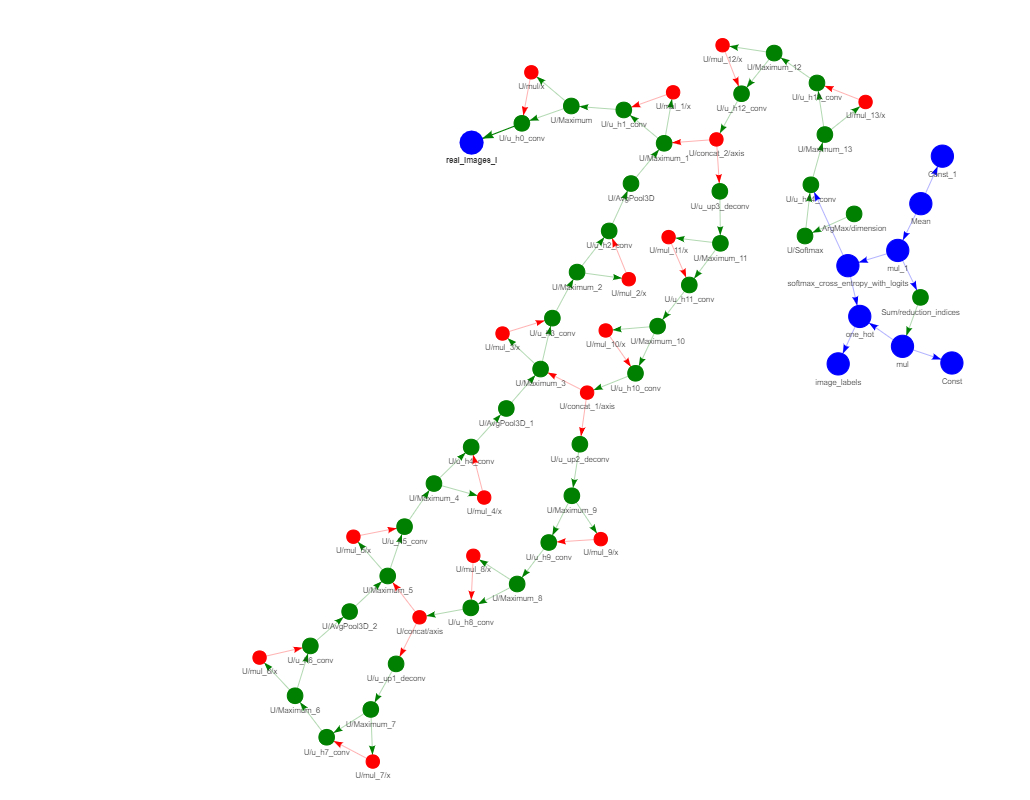
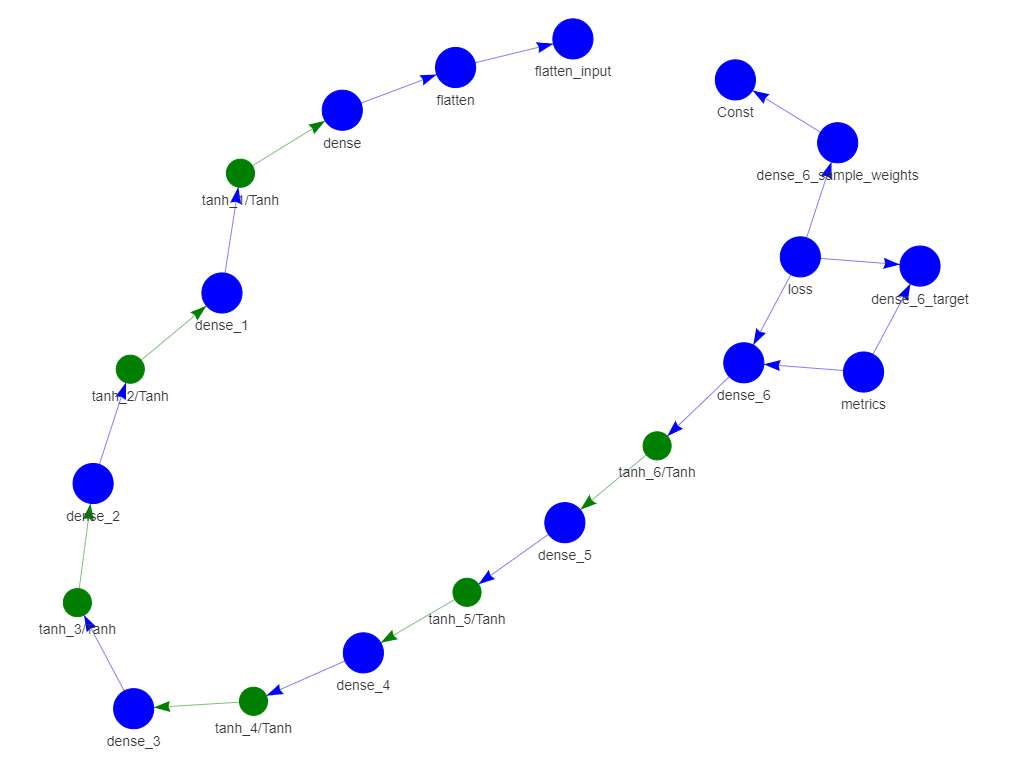


Figure 16: Simplified RDF graph for the MNIST fashion classifier.

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 the 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.

Figure 17: Samplified 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 ).



#### Test on real scientific paper’s code repository

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.

Table 1: Statistics of RDF graph extracted using code2graph.



## Light weight graph extraction

### 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.

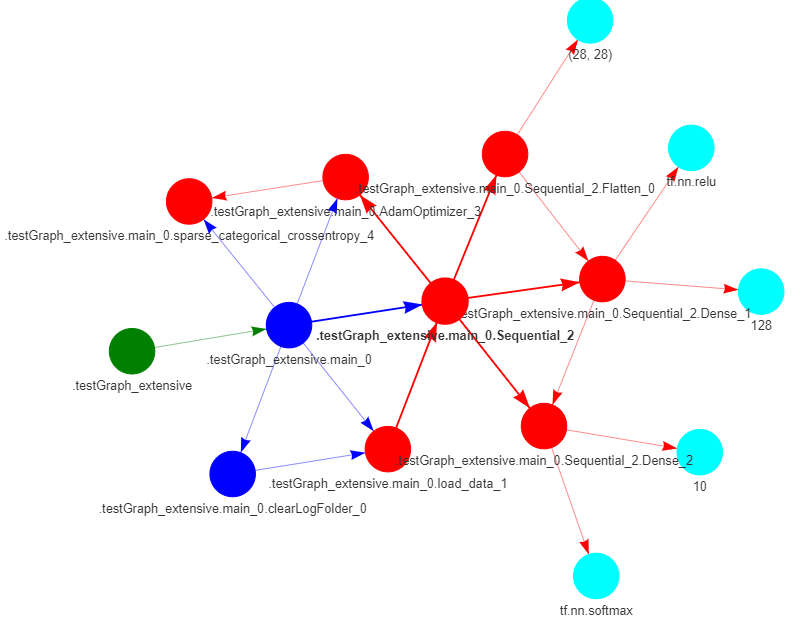


Figure 18:The visualization of the Lightweight RDF knowledge graph.

### Test on complicated graph

In the following figures, we also present results on more complex graphs.

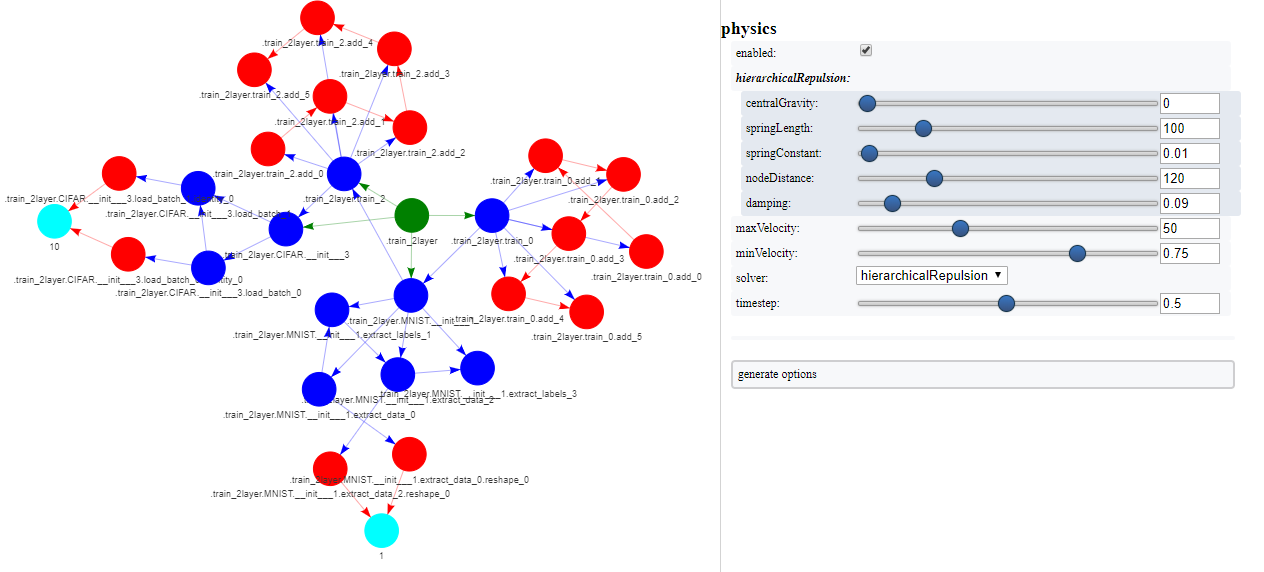


Figure 19: Light-weight AST-based graph extraction for complicated graph.

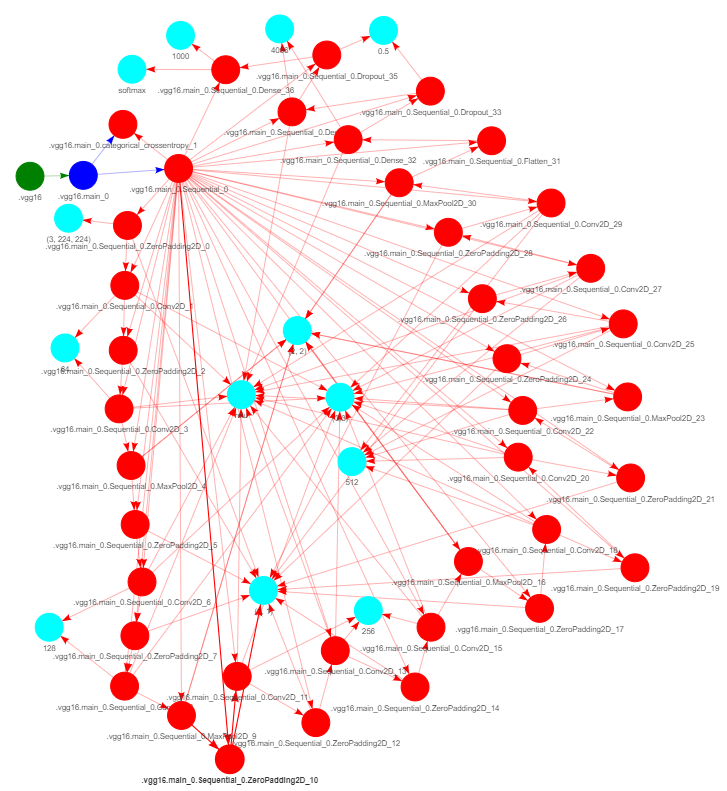


Figure 20:VGG16 RDF extraction

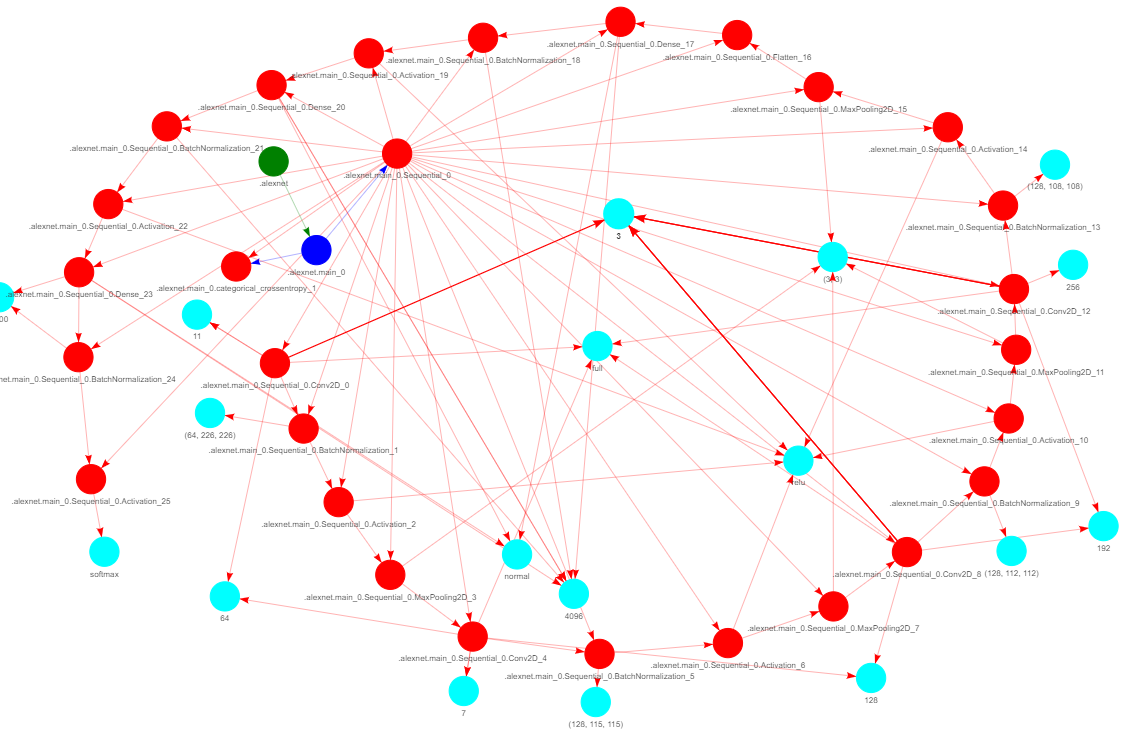


Figure 21: RDF graph extraction from Alexnet

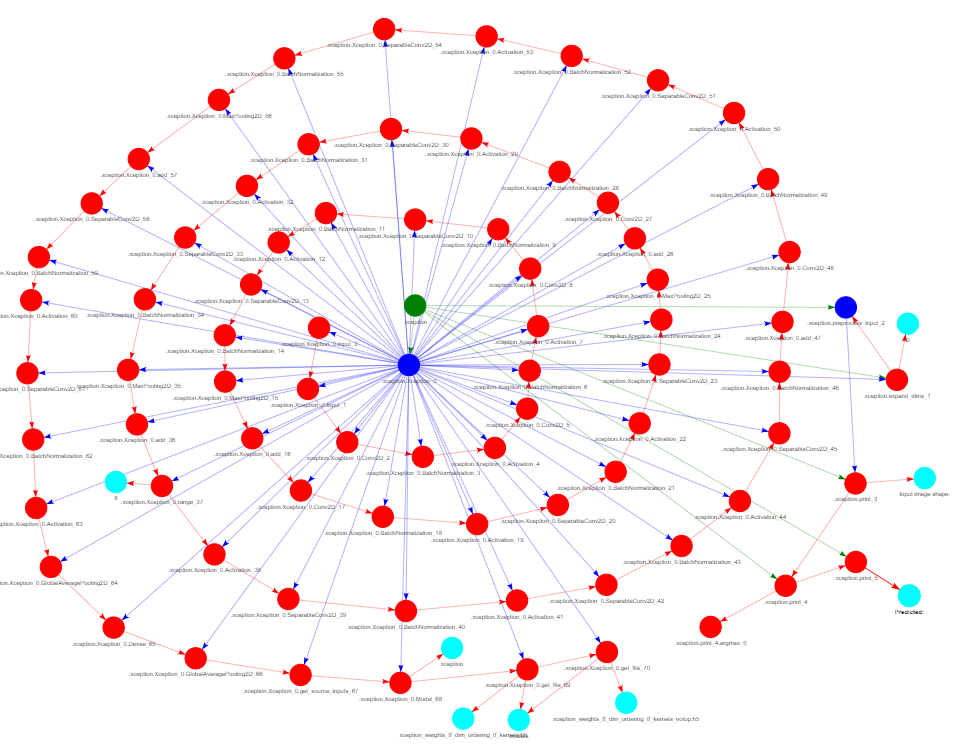


Figure 22: RDF extraction from Xception[[7]](#footnote-8)

## Future Work

Our future for Phase 2 will focus on the following directions.

### 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.

### Generating code from RDF graph



Figure 23: Pipeline for converting the RDF graph to Code

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 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:

### 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 the TensorFlow using the dataflow graph. Moreover, the suggested entities of the nodes maybe erroneous and may not exists 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.

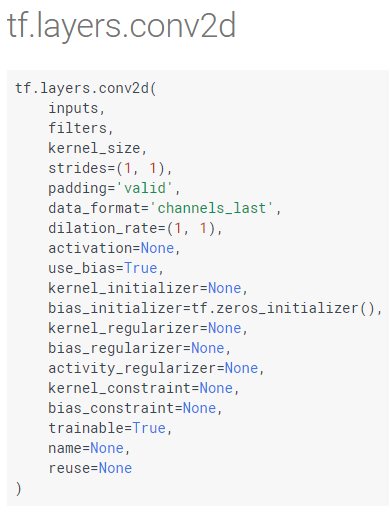


Figure 24: 2D convolutional lower level API's arguments.

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

Once the RDF graph has been filtered, we have to add the context and attribute necessary for executing the computational graph corresponding to the RDF graph. For example, as shown in Figure 24, the 2D convolutional API has various arguments whose attribute 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, 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 end of being just 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. Hence, 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.

### 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 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.

### Evaluation of the Graph2Code:

The evaluation of the 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 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 has been calculated using the computational graph generated from the inferred RDF 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 (see an example below).

**g = tf.Graph()**

**With g.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.

### Knowledge graph-embedding



Figure 28: Knowledge graph embedding using auto-encoder

In phase 2, the focus of the project will be 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 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, 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.

### Similarity measure and novelty measure in embedding space



Figure 29: Similarity and difference detection in embedding space.

After embedding the knowledge graph triples, and the larger part of knowledge graph, in second phase, we will perform the similarity and difference measurement utilizing the embedding values. The architecture will consist of utilizing the results of unsupervised models such as knowledge graph auto-encoder along with other supervised algorithms to acquire better measure of similarity and difference among the knowledge graph of scientific papers.

## 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 few. While the AST based approach that acquire 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, it requires partial compilation.

**References**

1. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012.
2. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
3. He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.
4. Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang et al. "Imagenet large scale visual recognition challenge." International Journal of Computer Vision 115, no. 3 (2015): 211-252.
5. Sethi, Akshay, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. "DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers." arXiv preprint arXiv:1711.03543 (2017).
6. Clark, Christopher, and Santosh Divvala. "PDFFigures 2.0: Mining figures from research papers." In Digital Libraries (JCDL), 2016 IEEE/ACM Joint Conference on, pp. 143-152. IEEE, 2016.
7. Wan, Yao, et al. "Improving automatic source code summarization via deep reinforcement learning." *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018.
8. Allamanis, Miltiadis, et al. "Learning natural coding conventions." *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.
9. Allamanis, Miltiadis, et al. "A survey of machine learning for big code and naturalness." *ACM Computing Surveys (CSUR)*51.4 (2018): 81.
10. Andrychowicz, Marcin, and Karol Kurach. "Learning efficient algorithms with hierarchical attentive memory." *arXiv preprint arXiv:1602.03218* (2016).
11. Alon, Uri, Omer Levy, and Eran Yahav. "code2seq: Generating sequences from structured representations of code." *arXiv preprint arXiv:1808.01400* (2018).
12. Allamanis, Miltiadis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to represent programs with graphs." *arXiv preprint arXiv:1711.00740* (2017).
13. Cvitkovic, Milan, Badal Singh, and Anima Anandkumar. "Deep learning on code with an unbounded vocabulary." *Machine Learning* 4.
14. Ben-Nun, Tal, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics." *arXiv preprint arXiv:1806.07336* (2018).
15. [Online]. Available: <https://github.com/zziz/pwc>.
16. [Online]. Available: <https://paperswithcode.com/>
17. [Online]. https://docs.python.org/3/library/ast.html
18. [Online]. <https://github.com/davidfraser/pyan>
19. Yi Luan, et al., "Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction," in *EMNLP*, 2018.

1. http://rat.nlplab.org [↑](#footnote-ref-2)
2. http://brat.nlplab.org/ [↑](#footnote-ref-3)
3. https://spacy.io [↑](#footnote-ref-4)
4. This means that each feature has a 1/3 likelihood of being dropped from the network. [↑](#footnote-ref-5)
5. https://scienceie.github.io/ [↑](#footnote-ref-6)
6. https://pypi.org/pypi/[package name]/json [↑](#footnote-ref-7)
7. <https://arxiv.org/pdf/1610.02357.pdf> [↑](#footnote-ref-8)