**DARPA ASKE DCC – Milestone 2, 2019**

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

Table of Contents

[1. Introduction 4](#_Toc536794579)

[2. Architecture Recap 4](#_Toc536794580)

[3. Datasets 5](#_Toc536794581)

[4. Representation 5](#_Toc536794582)

[5. Text2Graph 6](#_Toc536794583)

[6. Image2Graph 9](#_Toc536794584)

[6.1 Extract figures from research paper 9](#_Toc536794585)

[6.2 Figure classification 9](#_Toc536794586)

[6.2.1 Binary Figure Type Classification 10](#_Toc536794587)

[6.2.2 Multi-class Figure Type Classification 11](#_Toc536794588)

[6.3 DL architecture diagram analysis: 11](#_Toc536794589)

[6.4 Results 14](#_Toc536794590)

[6.4.1 Dataset Preparation: 14](#_Toc536794591)

[6.4.2 Figure Type Classification Accuracy: 15](#_Toc536794592)

[7. Code2Graph 17](#_Toc536794593)

[7.1 Pre-processing and environment re-creation 17](#_Toc536794594)

[7.1.1 Requirement installation 17](#_Toc536794595)

[7.1.2 Detect the main portion of the code 17](#_Toc536794596)

[7.1.3 Checking if the framework consists of just Tensorflow or Keras with Tensorflow 18](#_Toc536794597)

[7.2 Extracting the dataflow summary from the codes: 19](#_Toc536794598)

[7.3 Extracting the tf.GraphDef() from a given summary file: 20](#_Toc536794599)

[7.4 Converting tf.GraphDef() to JSON 20](#_Toc536794600)

[7.5 Ontologies for code2graph 21](#_Toc536794601)

[7.6 Converting Json to RDF graph: 23](#_Toc536794602)

[7.7 Test on real scientific paper’s code repository 29](#_Toc536794603)

[7.8 Implementation Details: 29](#_Toc536794604)

[7.9 Running the TestScript 31](#_Toc536794605)

[7.9.1 Install the requirements 31](#_Toc536794606)

[7.9.2 Run the Jupyter Notebook 31](#_Toc536794607)

[8 Future Work 33](#_Toc536794608)

[9 Potential Papers 34](#_Toc536794609)

[Works Cited 34](#_Toc536794610)

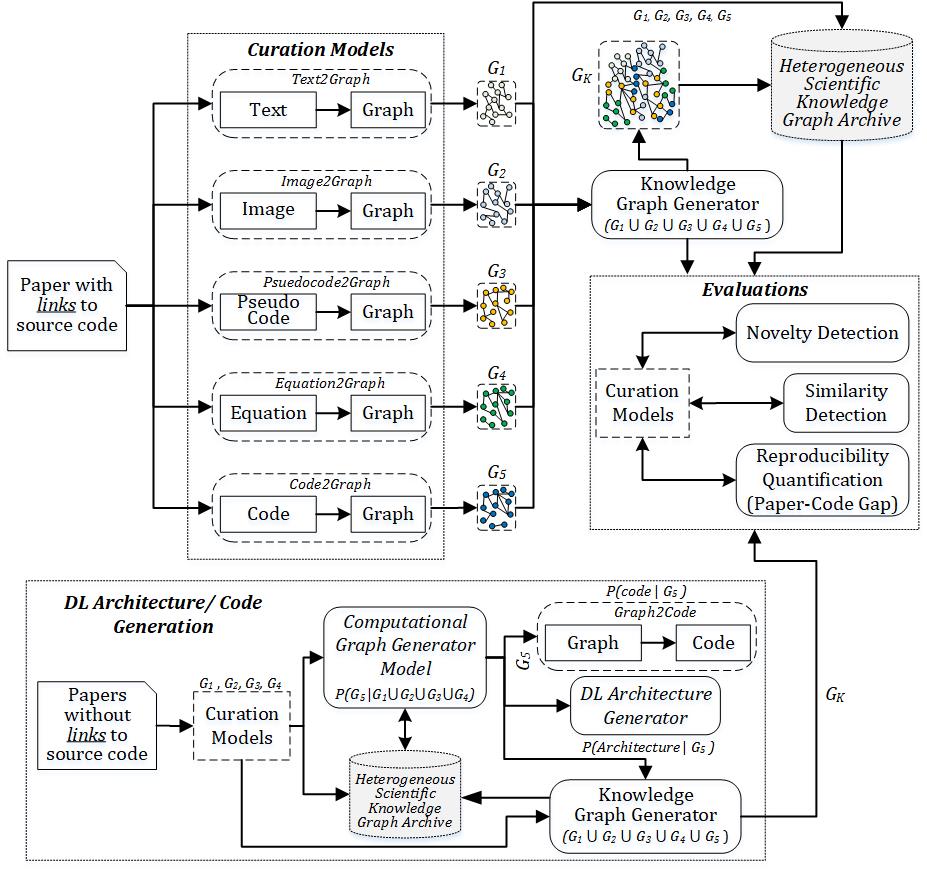
[Recurrent Neural Networks for NER 35](#_Toc536794611)

[Conditional Random Fields 37](#_Toc536794612)

# 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.

# 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 embarked on the task of curating Deep Learning papers, their associated code and implementation details. To aid in this we cited three sources – (paperswithcode [1], pwc [2] and GitXiv [3]). Out of the three sources, paperswithcode had the most structured curation followed by pwc. GitXiv, has a larger collection of papers, but they are fragmented and not all point to valid scientific content. Some of them point to blogs and just implementation details. So as a starting point we used the following two sources detailed below:

|  |  |  |
| --- | --- | --- |
| Source | Paper # | Implementations (Yes/No) |
| Papers with code[[1]](#footnote-2) | 100 | Yes |
| PWC[[2]](#footnote-3) | 69 | Yes |

# Representation

We use knowledge graphs as our representational model. Knowledge graphs link uniquely identifiable web resources to other such web resources or literal values by means of relationship edges. Every unique node-edge-node in the graph can be represented as a Subject-Predicate-Object (SPO) triple. Each of the graph components focuses on extracting information in the form of such subject-predicate-object triples. This representation has not changed from the last report. The only update is that we have realized that to represent program graphs we would need to be able to represent sequences in knowledge graphs because code blocks are executed as sequences. For this specific purpose we have decided to include SHACL [4] constraints in RDF knowledge graphs. SHACL (SHApes Constraints Language), is a constraint on top of RDF graphs that allows RDF graphs to have a shape and represent the order in which shapes occur. This will be useful as we progress to representing code sequences and algorithm sequences.

# Text2Graph

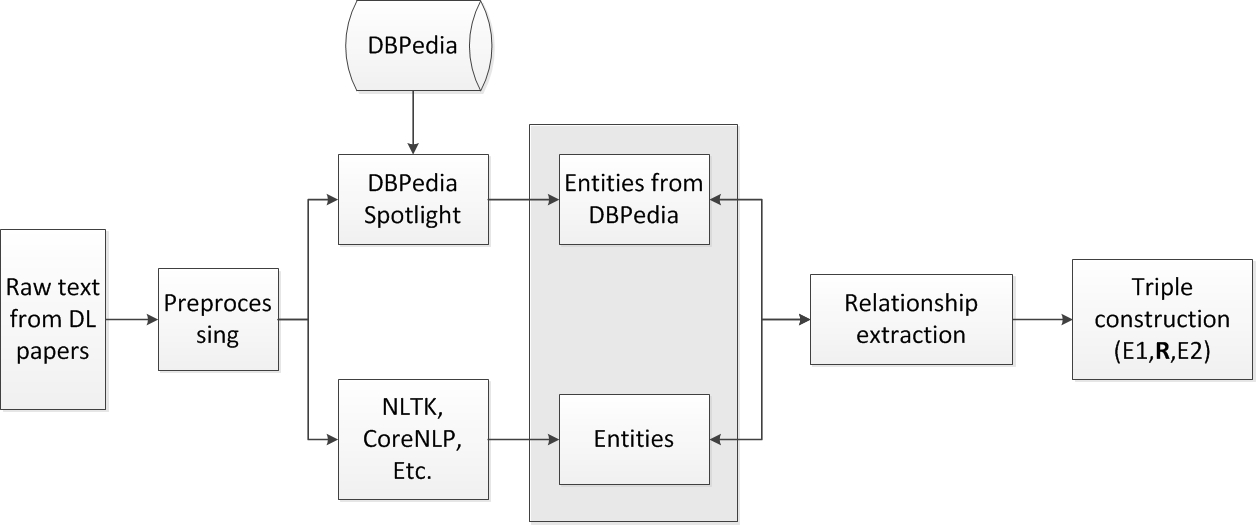


Figure 1: Outline of the text2graph pipeline

Figure 1shows the outline of our text to graph framework. The framework first detects and annotates the scientific entities present in the text through a Named Entity Recognition (NER) process. The entities thus generated are used in a related task called Relation Extraction. This task accomplishes the end goal of Phase I - to generate triples representing scientific facts from text. These triple statements have the form {*subject, predicate, object*}. The subject and object components are extracted by NER. The predicate on the other hand is the ‘glue’ specifying the relationship between two entities. It is identified by relationship extraction. Thus, we can view the triple statement as a {*entity1, relationship, entity2*}. The graph generation from text goes through the sequence of tasks described below.

**Dataset preparation:**

All the datasets used consisted of pdf documents. Extracting text from PDF documents is a challenge. We used the open source python library *pdfminer*[[3]](#footnote-4) to extract text from PDF documents. We then proceed to clean the pdf documents to accurately represent the scientific text. The image extraction process is detailed in the image2graph section. Since our focus is on figuring out the reproducible aspects of a scientific document, we start by looking for contributions and the novel aspects that each paper addresses. We take cue from the structure of scientific papers and use the headings as our starting point to find relevant text. For the purposes of this milestone we have extracted content from abstracts of all the documents. In the next iteration we will be focusing on {Abstracts, Contributions, Results, Conclusions}. The extracted sections are converted to sentences for the annotation process. The statistics are shown in the following table.

|  |  |
| --- | --- |
| Paper Source dataset | paperswithcode |
| Total number of abstracts extracted | 100 |
| Total number of sentences | 30,524 sentences |

**Named Entity Annotation**

The Named Entity Annotation process included the following two approaches:

**Dbpedia-spotlight:**

The first approach relied on improving the entity annotation models provided by dbpedia-spotlight[[4]](#footnote-5). This tool provides a convenient annotation to link entities to large-scale knowledge graphs like DBpedia. Since our goal was also to utilize the hierarchies present in such large-scale graphs, we annotated the abstract sentences using dbpedia-spotlight.

We improved the dbpedia-spotlight model to pick annotations that are more context specific. For example, in the text, *We propose new methods for NMT..*, the tool annotates *NMT* as [*http://dbpedia.org/resource/Nordic\_Mobile\_Telephone*](http://dbpedia.org/resource/Nordic_Mobile_Telephone)*.*  However, NMT in this specific context refers to *neural machine translation.* On further inspection, this showed that dbpedia has not yet curated the page for *neural machine translation.* We improved this process to extend dbpedia annotations for more models.

**CONLL Annotation**

The entities annotated by DBpedia were incompatible with the annotation scheme for most relation extraction tasks. To build a quick scientific graph generator, we relied on the entities generated from the DBpedia annotation and used a distant supervision technique to annotate the entities. This approach is commonly used for sequence tagging problems. Each line represents a single word with a series of tab separated fields. The most recent version uses 12 columns:

**ID FORM LEMMA PLEMMA POS PPOS FEAT HEAD PHEAD DEPREL PDEPREL**

In general, because this is a shared task, not all the columns are required. The columns that we use for our annotation scheme are:

**ID** - The sentence id for the word. This starts at 1

**FORM** – The word itself

**POS** – The part of speech of the word

**TAG** – The tag for entities or relationships that annotates the word. If the word is part of a multi word entity, we use the BIO scheme to annotate the named entity, where B refers to begin, I to inner and O to out. For example, the word *model* is annotated as **B-Model** and the word *deep learning model* can be annotated as **B**-deep **I**- learning and **O**-model.

**Named Entity Extraction**

After the annotation we built a Bidirectional LSTM (BiLSTM) with a CRF on top. This model has been shown to provide state-of-the-art results [5]. Utilizing past work [5], we annotated the dataset and identified six types of entities:

* **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.

In addition, we also annotated the dataset with eight types of relationships:

* **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 [5], 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.

# Image2Graph

In this section we present our novel end-to-end framework that automatically locates all figures in 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 our initial results. Apart from the algorithms, we also created a labeled dataset to train the classifiers and test their efficacy.

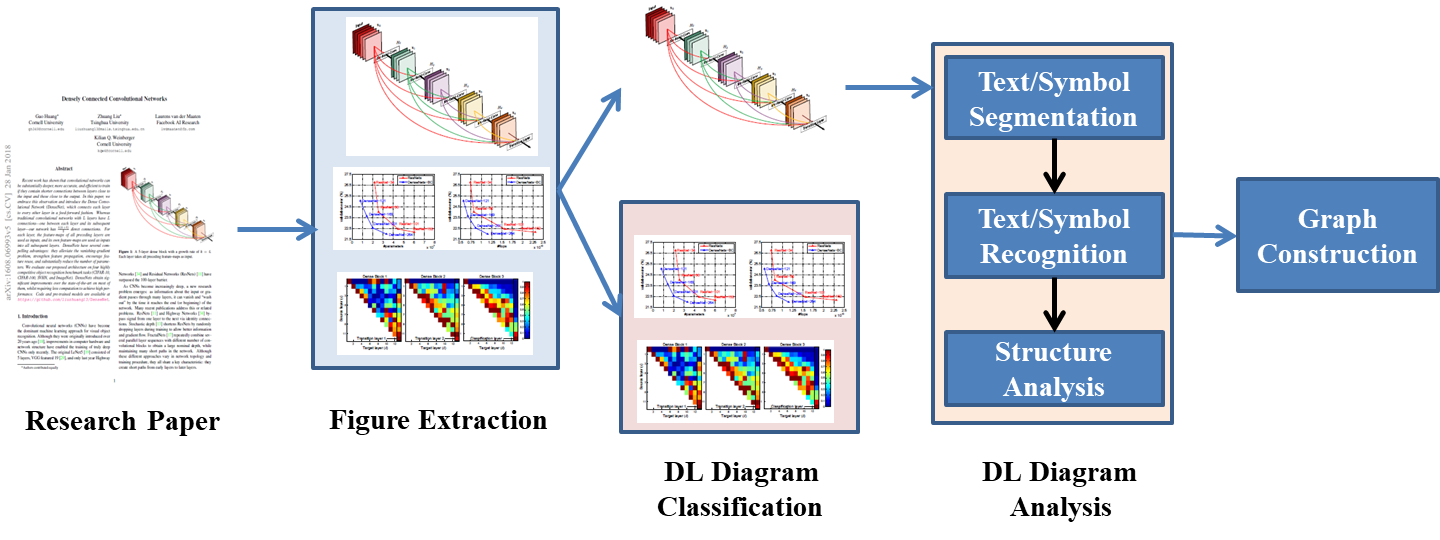
****

Figure 2: The architecture of the image2graph module

## 6.1 Extract figures from research paper

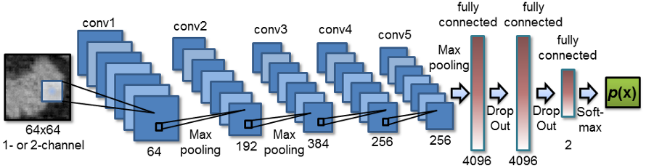
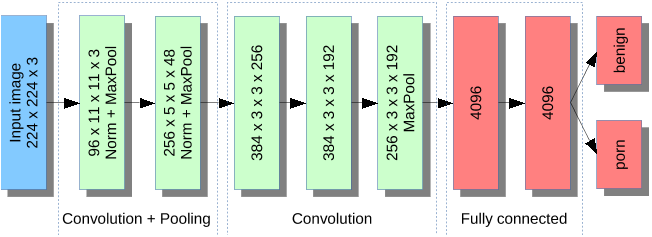
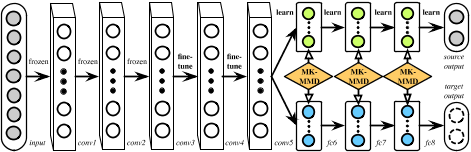
We used PDFFigures 2.04 [6] tool to automatically extract a set of figures from a research paper. We downloaded 1000 papers from arXiv.org using “deep learning” as the input query. 8310 figures were extracted from these 1000 downloaded papers.

## 6.2 Figure classification

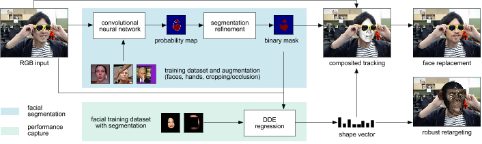
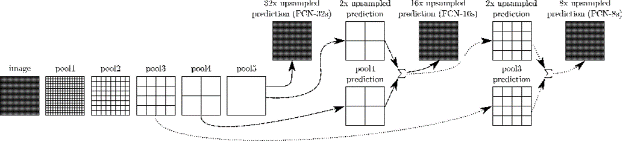
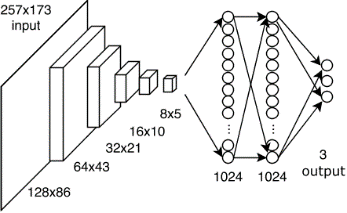
We noticed that the figures from deep learning papers follow different patterns/styles. Figure classification is performed in two steps. First, only the DL diagrams are identified and retrieved from all the figures and tables extracted from a research paper. We develop a binary classifier to perform this task.

It has been observed that the DL architecture diagrams typically do not follow any definition and show extreme variations. Sethi et al. [7] divided model diagrams into 5 broad categories 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, and (v) Pipeline plot: along with the DL model design, the entire pipeline and mostly some intermediate results of image/ text is shown as well.

However, we found that this categorization does not include matrix box plot. So, we introduced another class called “Matrix Box” as shown in Figure 3. It is essential to account for these variations in the proposed pipeline, as they indicate the DL design represented in the paper. Following this assumption, the proposed approach does not identify a DL design flow that is not represented in a figure. Next, we train a fine-grained six-class classifier to classify the figure into one of the six broad categories as follows.



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

### 6.2.1 Binary Figure Type Classification

The aim is to classify the extracted figures into two classes, namely DL architecture diagrams or others (e.g. plots, tables, photos, etc.). We developed a binary neural network classifier trained on fc2 features to perform the prediction using the architecture shape and the flow that has different characteristics compared to a result graph, or sample image. We employed three network architectures: VGG16, VGG19 [8] and ResNet-50 [9]. All the three networks are pre-trained on the 1.2 million images from ImageNet [10] and then fine-tuned for our figure classification task.

All the figures are resized to 224×224 for VGG models, and to 299×299 for ResNet model. Employing the pre-trained models, we extract 4096 convolutional features (fc2) from the input images. On the extracted features (CNN codes), a small fully connected binary NNet classifier with two hidden layers of size [1024, 256] is applied with batch normalization and drop out to prevent overfitting as shown in Figure 4.

Since the size of the training data set is small, we employed data augmentation technique to increase the accuracy of the classifier. Data Augmentation alters our training batches by applying random rotations, cropping, flipping, shifting, shearing etc. In the specific dataset, random cropping does not make sense. So, we added horizontal flipping and random shifting up and down and side by side because all these scenarios are likely.

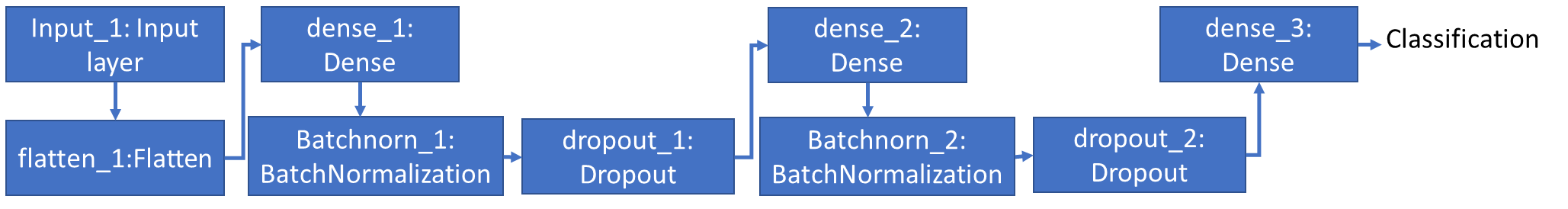


Figure 4: Deep neural network architecture for binary figure classification

### 6.2.2 Multi-class Figure Type Classification

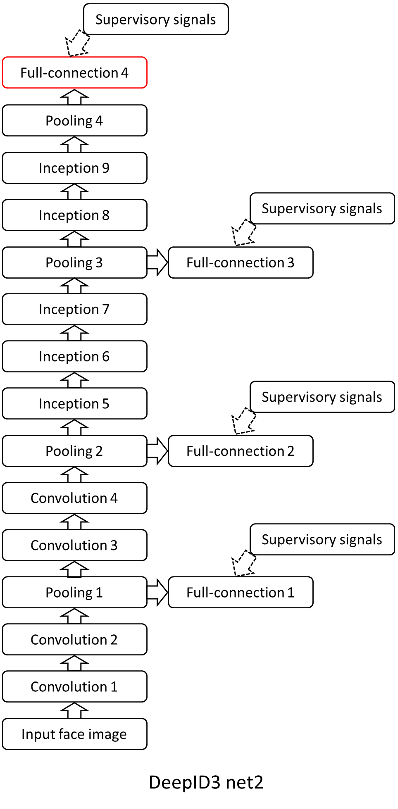
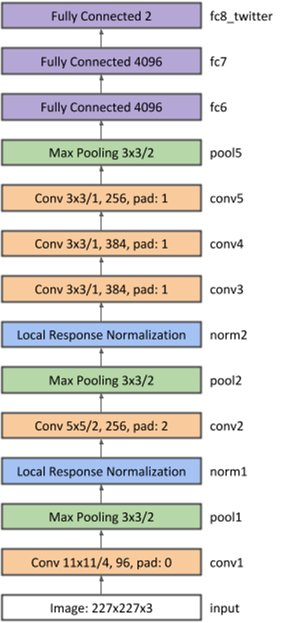
Once the DL figures are identified automatically, we plan to employ another six-class neural network classifier to identify the category of the input DL architecture figure. We follow similar steps as described for binary classifier to extract features from three pretrained models. Next, a six-class classifier was trained with one hidden layer of size 1024. Having a sequence of two classifiers provided better performance as compared to a single classifier with seven classes (seventh class being no DL design flow). As data augmentation was used to train this model, it can also handle slight variations in the images such as horizontal flip, different illuminations, rotations and shifting up and down which are likely to present in diagram images.

## 6.3 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 5, 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 5 (a).

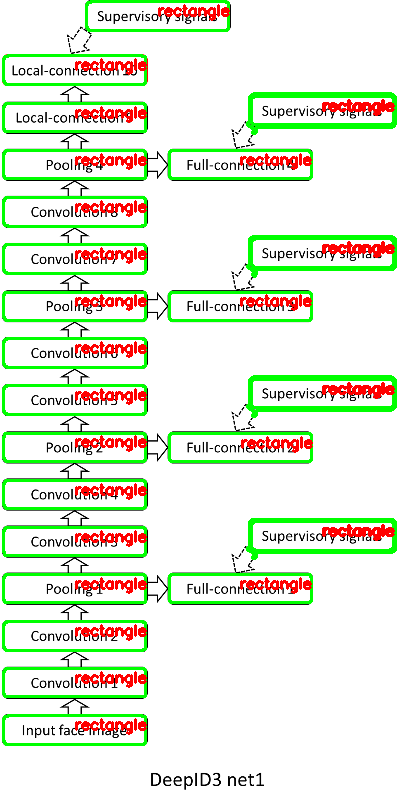
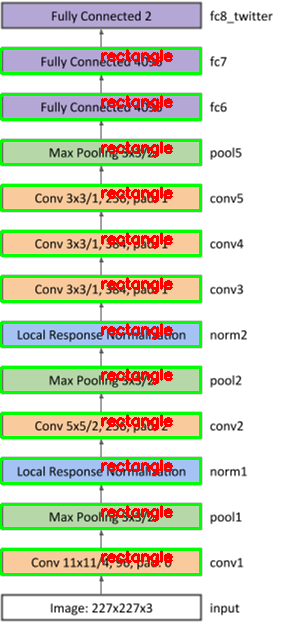
 

(a) (b)

Figure 5: DL 2D diagram (1) line diagram, (b) block diagram

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 node 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 6(b).

In the next stage, we will improve the node detection method for different types of DL diagrams and develop algorithm to detect arrows present in DL diagrams. The direction of the arrows will give important information about the flow of the entire DL model design. This will link to the graphs produced by text2graph and code2graph.

1. (b)

Figure 6: Node/Layer Detection

**Text Detection:**

Once the nodes are extracted, the text in each node/layer is obtained through OCR using Tesseract7. Based on our manual observation, we assume that a layer description is present near the detected node (either inside or in nearby region). The results of text detection from four diagrams are using conventional Tesseract is shown in the Figure 7. Note that the *red* text shows the output of the text detection algorithm from four DL diagrams. However, the accuracy of the current algorithm can be improved by incorporating a dictionary of possible DL layer names. We plan to use this dictionary to perform spell correction of the extracted OCR text.

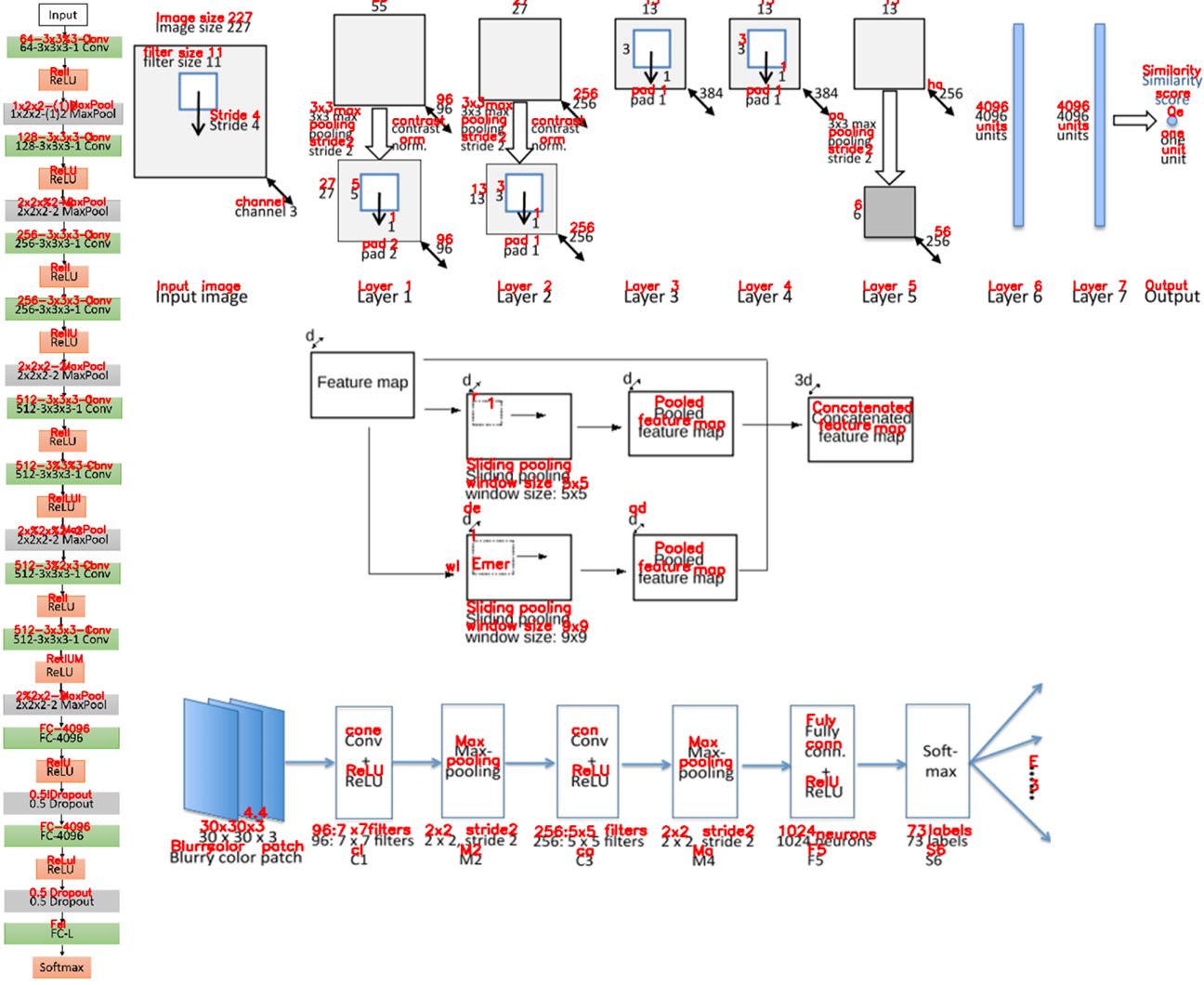


Figure 7: Results for text recognition

## 6.4 Results

### 6.4.1 Dataset Preparation:

As mentioned earlier, we downloaded 1000 papers from arXiv.org to extract 8310 figures from them. Next these figured were labeled for training the classifiers.

**Binary classification labeled dataset:**

Out of the 8310 extracted figures, we manually identified which ones did not contain a DL design flow. These represent the usual figures that are found in a deep learning research paper that does not contain a design flow. There were 1305 DL design flow diagrams and 7005 other figures.

**Multiclass classification labeled dataset:** Further, to evaluate the fine level, five-class, figure type classification, the DL design flow diagrams were manually labelled. The distribution of figures are as follows: (i) Neurons plot: 80, (ii) 2D box: 343, (iii) Stacked2D box: 185, (iv) 3D box: 246, (v) matric box: 56, and (vi) Pipeline plot: 286.

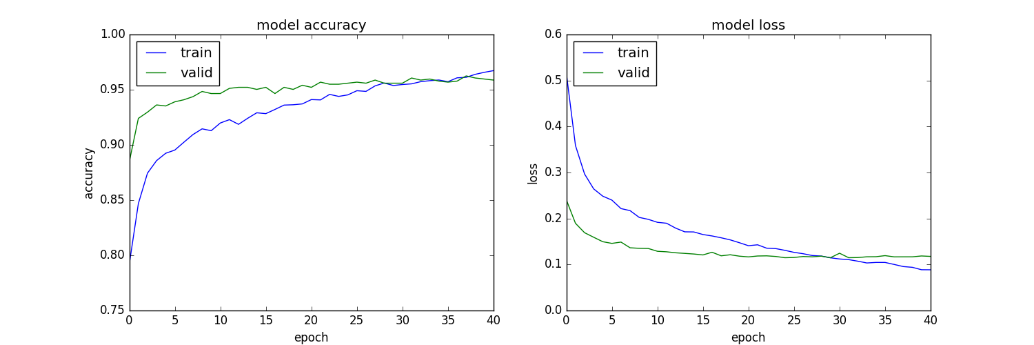


Figure 8: The accuracy/loss graph for binary classifier using VGG 19 features

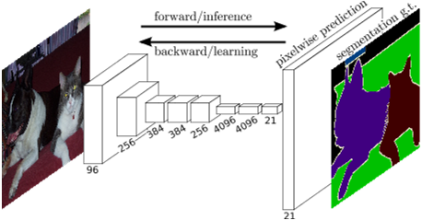
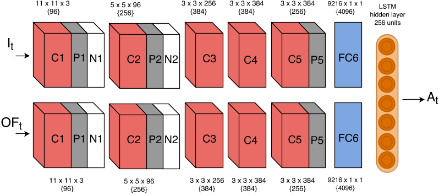
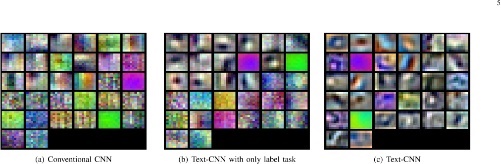
### 6.4.2 Figure Type Classification Accuracy:

To evaluate the coarse level binary classification, the whole dataset is split as 80% for training, 10% validation, and 10% for testing. For each experiment, only the best models were saved along with their weights (a model only gets saved per epoch if it shows higher validation accuracy than the previous epoch). The accuracy/loss graph of the network using VGG19 features is shown in Figure 8. The results are computed for three different classifiers as shown in Table 1. It can be observed that the DL diagrams can be identified with 95.5% accuracy on the test dataset which is noticeably better than the 86.2% accuracy reported in [7]. This model is quite robust as it has similar performance on the validation dataset and the test dataset.

**Table 1:** The performance of three binary classifiers

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Architecture | Accuracy (%) | | | Loss | | | F1 Score (%) | | |
| Train | Validation | Test | Train | Validation | Test | Train | Validation | Test |
| Vgg16 | 96.00 | 95.50 | 95.03 | 0.098 | 0.113 | 0.147 | 96 | 96 | 95 |
| Vgg19 | 96.73 | 95.87 | 95.49 | 0.089 | 0.118 | 0.156 | 97 | 96 | 96 |
| ResNet | 92.23 | 91.51 | 90.99 | 0.165 | 0.223 | 0.359 | 92 | 92 | 91 |

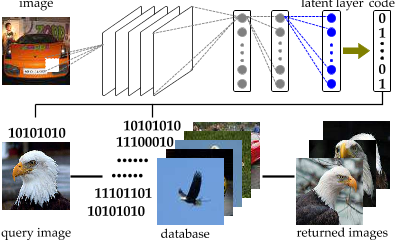
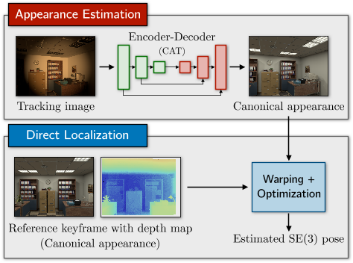
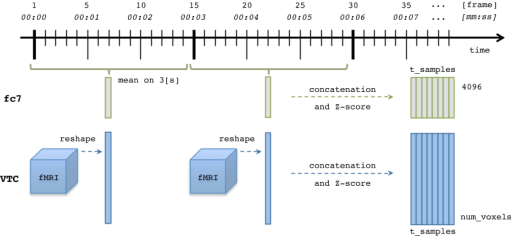
**Randomly sampled correct labels:**

True Label: Architecture True Label: Architecture True Label: Other

Predicted Label: Architecture Predicted Label: Architecture Predicted Label: Other

**Randomly sampled incorrect labels:**

True Label: Architecture True Label: Architecture True Label: Other

Predicted Label: Other Predicted Label: Other Predicted Label: Architecture

It appeared the model predicted “other” to most of the incorrect “architecture” images which are the dominant classes in the provided training set. This problem can be overcome by adding more data for the other classes, either via data augmentation or by collecting figure data from more papers.

In case of the multi-class classifier, 80 - 10 - 10 train, validation, and test split is performed to train the classifier for six class classification. The results are shown in Table 2. It can be observed that even on highly varying DL flow design images, identifying the type of DL flow is more than 70% accurate.

**Table 2:** The performance of three multi-class classifiers

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Architecture | Accuracy (%) | | | Loss | | | F1 Score (%) | | |
| Train | Validation | Test | Train | Validation | Test | Train | Validation | Test |
| Vgg16 | 92.98 | 71.42 | 61.22 | 1.94 | 2.38 | 2.95 | 93 | 72 | 63 |
| Vgg19 | 89.75 | 62.24 | 60.18 | 1.89 | 1.05 | 3.10 | 90 | 62 | 60 |
| ResNet | 86.19 | 67.34 | 58.63 | 1.93 | 0.936 | 2.88 | 87 | 68 | 59 |

# Code2Graph

In this section we provide the implementation detail for extracting the RDF knowledge graph from the code repositories of the scientific papers. The proposed pipeline/architecture for extracting the RDF graph from the code consists of the following stages (shown in Figure 9):



Figure 9: Pipeline for extracting the RDF graph.

## 7.1 Pre-processing and environment re-creation

The first task in extracting the knowledge graph consists of pre-processing and re-creating the code environment in which the original code was executed. To train our model we have been provided a dataset by Siemens. This dataset contains published papers that have their codes available to the public. We found datasets like zziz [2]and paperswithcode [1], however, the problem with these datasets is that they have code using frameworks other than TensorFlow or code unrelated to machine learning. We first eliminated the ones with other frameworks, since our focus will be on TensorFlow or Keras with Tensorflow as backend only. The code in the dataset have the architectural data that we are mostly interested in. To extract this data we have tweaked the codes to get their event.summary files. These files include the computational graph that the model runs. It contains every piece of information to build the model. Thus, these files include everything we need to define the architecture of the model. The implementation details and steps for performing this are as follows:

### 7.1.1 Requirement installation

This step is necessary before we can build the deep learning model and extract the computation graph. In order to recreate the code environment, we check for the required python version, install required libraries, and make sure we can run the code. Although some efforts have been given in automating this step, most of it is still done manually.

### 7.1.2 Detect the main portion of the code

The main portion or starting root of the code can exists in different files. In our approach, we search for the main.py file or <'name\_of\_the\_algorithm >.py. Sometimes, the train.py also exists, and may contain the main code. In most of the repositories we found a model class that contains all the information necessary to build the model and a main file that creates a model object, adds some attributes to it and eventually compiles the model or runs the TensorFlow session. In this case, we find where the model is compiled (for Keras) or find where the session (for Tensorflow). In some cases, the code imports the architecture from other compile python libraries. Hence, we carefully extract the architecture (which may be improved version of the previous architecture) that is relevant to the current scientific paper rather than the old architectures.

### 7.1.3 Checking if the framework consists of just Tensorflow or Keras with Tensorflow

We check if the code consists of session definition of model compilation information. If it uses TensorFlow, we locate where the session is defined. After that, we find where the model definition is ends. Then we add the following lines before the training starts.

1. import tensorflow as tf
2. import sys
3. writer = tf.summary.FileWriter("../tmp/log/", sess.graph)
4. sys.exit()

where 'sess' is the current TensorFlow session that was started.

If the code uses Keras, then first we get the session and then dump its default graph to an event.summary file like in the TensorFlow's case. We add the following lines after the model.compile(), and before the model.train() to achieve this. It will first get the session since the sessions in Keras are created implicitly, then it will dump the summary file.

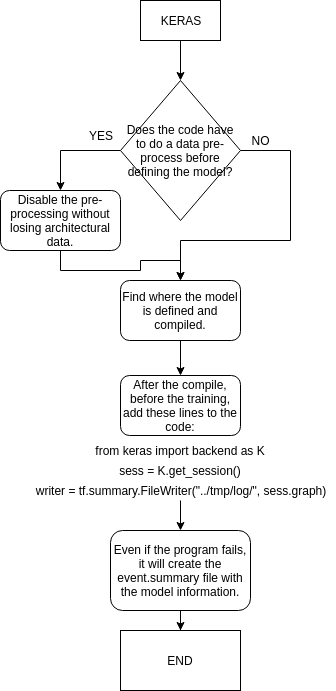
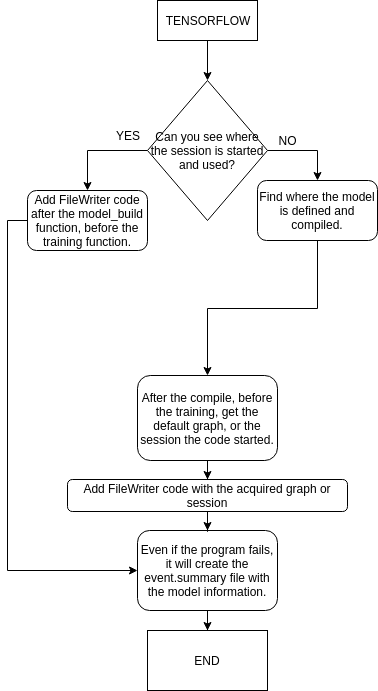
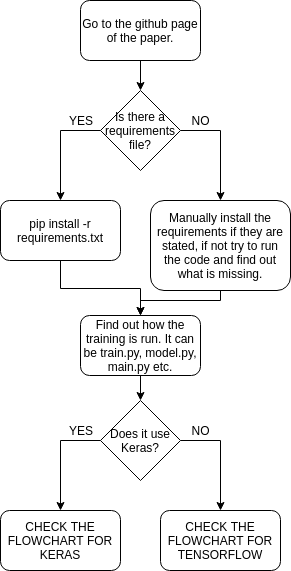
1. from keras import backend as K
2. import tensorflow as tf
3. import sys
4. sess = K.get\_session()
5. writer = tf.summary.FileWriter("../tmp/log/", sess.graph)
6. sys.exit()

These lines must be added before the training starts. If the training phase begins, it will take a long time for the program to be completed. In some cases, code authors get the input size from the given data. In such case we refer to the code repository’s Readme file and find an example data they have used. Then the input size is analyzed, and the code is changed accordingly. i.e. we change the variable input and output sizes to some constants. Only then the model compilation will be successful.

The flowcharts (shown in Figure 10) summarize the above pre-processing steps.

Automating this process is unfortunately non-trivial since the extraction requires some understanding of the architecture. It must be done manually.

Figure 10 Flowcharts Explaining the Event Summary Extraction



## 7.2 Extracting the dataflow summary from the codes:

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 [11], 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.

## 7.3 Extracting the tf.GraphDef() from a given summary file:

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 as follows:

1. from tensorboard.backend.event\_processing import plugin\_event\_accumulator as event\_accumulator

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

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

## 7.4 Converting tf.GraphDef() to JSON

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. Inorder to convert the graphDef to Json we have utilized a built in module called MessageToJson. Protocol buffer has a method built in called MessageToJson under json\_format. (google.protobuf.json\_format.MessageToJson(...)) The details of json\_format module can be found from [Google Protobuf Documentation website](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.json_format-module) at [12]. A glimpse of the function prototype is below:

1. def MessageToJson(message, including\_default\_value\_fields=False)
2. '''
3. Converts protobuf message to JSON format.
4. Args:
5. message: The protocol buffers message instance to serialize.
6. including\_default\_value\_fields: If True, singular primitive fields,
7. repeated fields, and map fields will always be serialized. If
8. False, only serialize non-empty fields. Singular message fields
9. and one of fields are not affected by this option.
10. Returns:
11. A string containing the JSON formatted protocol buffer message.

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

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

## 7.5 Ontologies for code2graph

In order to extract the hierarchical structure of the deep learning architecture we have extracted the ontology from the TensforFlow github repository [13] to guide us in separating the deep learning architecture from the computation graph metadata necessary for the TensorFlow core C/C++ runtime. TensorFlow has a well-defined structure which can aid in describing the deep learning architectures (such as a layer, loss, optimizer, input, output, etc.,). Although tf.GraphDef() will have these definitions by default, it will consists of extra computation graph information. Hence, to aid us on focusing on deep learning architecture we have extracted the TensorFlow API module hierarchy (seeFigure 11).

**A screenshot of a cell phone

Description automatically generated**

Figure 11: TensorFlow API Module hierarchy information used for ontology generation

As can be seen from Figure 11 that 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 deep learning 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 deep learning architecture present in the code.

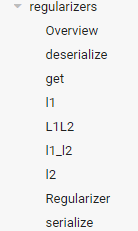
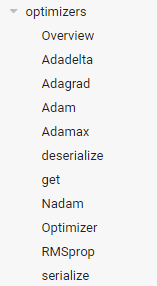
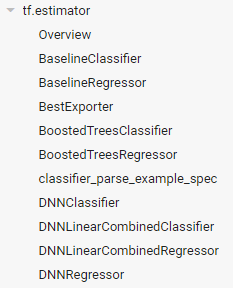
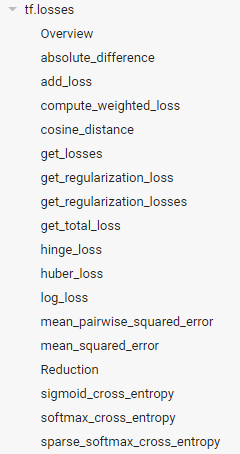
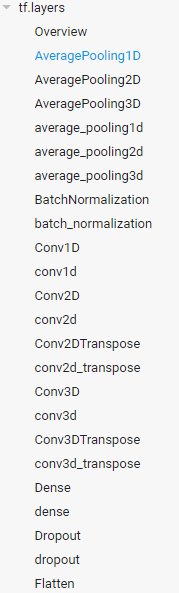


Figure 12: Some more examples on hierarchical information used for ontology generation

Some more examples on the hierarchical information of TensorFlow used in ontology generation are shown in Figure 12. Furthermore, the extracted ontology may also be visualized using networkx as shown in Figure 13.

A close up of a logo

Description automatically generated

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).

## 7.6 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 [14]. 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 type of nodes, virtual nodes. Using these nodes, allowed us to represent the architecture of deep learning algorithms. Although, it might be intuitive to assume that all the nodes should be connected to the Model (or Root), the complexity of the RDF graph will become intractable as the complexity of the deep learning algorithm increases. In addition, the extracted hierarchical information can be represented with different resolution (in terms of depth from the root node). Therefore, the first objective is to get the sub-components of this hierarchical structure and collapse these components as needed to determine the resolution of the extracted RDF graph.

Coming back to our 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:

1. {'attr': {'\_class': {'list': {'s':['...==']}},
2. 'dtype': {'type': 'DT\_FLOAT'}},
3. // => turned into a list of attr list
4. ['\_class', 'list', 's', '...==',
5. 'dtype', 'type', 'DT\_FLOAT']

A glimpse of the realized pipeline is shown below,

1. **for** node **in** nodes:
2. **if** 'attr' **in** node.keys():
3. attr\_paths = self.recursive\_scan\_attr(node['attr'], path=[])
5. sb = BNode(node['name'])
7. **for** attr\_path **in** attr\_paths:
8. **if** **not** sum([1 **if** rule.parse(attr\_path) **else** 0 **for** rule **in** self.rules]):
9. **print** ("can not recognize %s" % str(attr\_path))
10. **raise** Exception("Unknown attribute")
12. **for** rule **in** self.rules:
13. results = rule.get\_results(sb)
14. **if** results:
15. self.RDF\_graph += results

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

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, "...==">

A glimpse of the type rule generator is as below:

1. **class** DTypeGenerator(GGenerator):
3. **def** \_\_init\_\_(self):
4. self.attr\_uriref = URIRef("http://example.org/dtype")
5. self.value = None
6. self.parsed = False
8. **def** parse(self, attr\_list):
10. **if** attr\_list[0] == 'dtype':
11. self.value = attr\_list[2]
12. self.parsed = True
13. **return** True
14. **return** False
16. **def** get\_results(self, sb):
17. triples = []
19. **if** self.parsed:
20. triples.append((sb, self.attr\_uriref, Literal(self.value)))
21. self.clear()
23. **return** triples

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 (see Figure 14). 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.

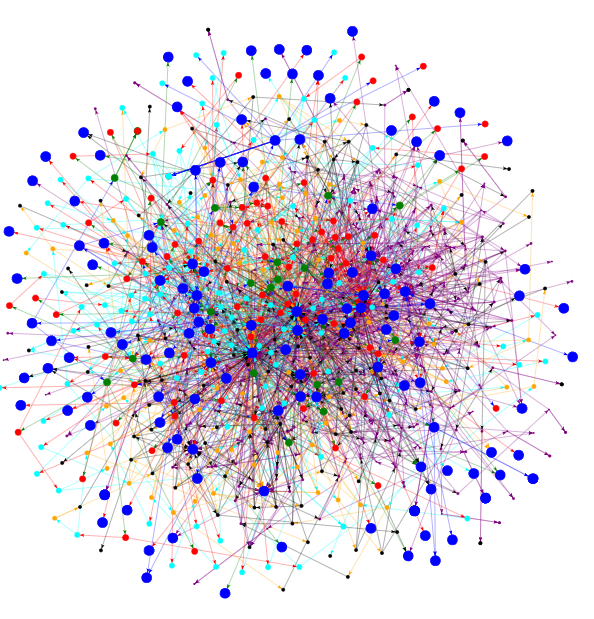


Figure 14: Visualization of RDF graph extracted from MNIST fashion classifier [15]

**Simplify the resultant RDF graph**

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

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. A screenshot of simplified RDF graph for MNIST fashion example is shown in Figure 15. To some extent, this simplified graph already can represent the deep learning architecture designed for the MNIST fashion example.

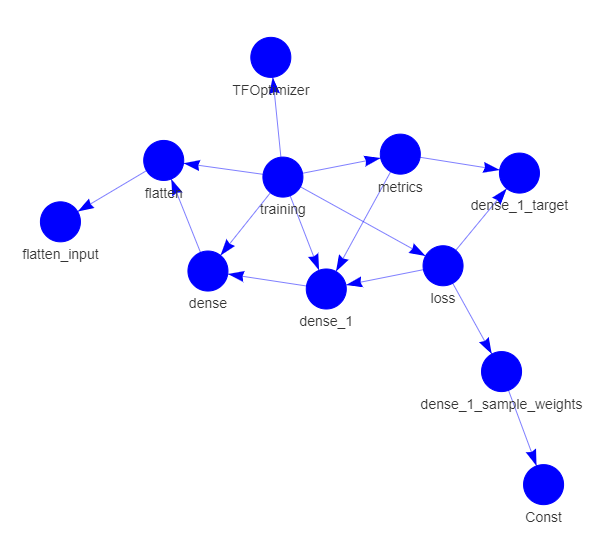


Figure 15: Simplified RDF graph for the MNIST fashion classifier.

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 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.

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

Table 3: Statistics of RDF graph extracted using code2graph.

****

After running the code2graph, the statistics for various attributes before and after simplifying the graph is presented in Table 3. 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.

## 7.8 Implementation Details:

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

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

Under graphHandler class, graphDef, RDF, and JSON formats are saved as properties.

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

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

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

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

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

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

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

1. **def** convertJson2RDF(self, jsonGraph=None):
2. '''''function to convert json to RDF format'''
3. **if** jsonGraph!=None:
4. self.\_jsonGraph=jsonGraph
6. self.\_RDF, self.\_sRDF = j2g.json2RDF(self.\_jsonGraph)
7. **return**

The Json2RDFParser class takes the extracted JSON string and parses it. It creates triples according to the RDF structure. All the steps from the previous chapter have been implemented as methods to this class. In the end json2RDF function creates a Json2RDFParser object and uses its parse method to call all the methods responsible for parsing the JSON string.

## 7.9 Running the TestScript

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

### 7.9.1 Install the requirements

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

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

### 7.9.2 Run the Jupyter Notebook

By running the following line Jupyter Notebook session is started:

1. **jupyter notebook**

This will open the browser with the notebook environment. From there navigate to testScript folder and open code2graph.ipynb file. This is the notebook (seen in Figure 16) we have prepared for test purposes. It is possible to run our code cell by cell. Run each cell from top to bottom by clicking on “Run”. The cell in the far bottom will display the RDF graph we have extracted from the given event file. It will both create a visualization in a new browser page and print out each triple that has been added to the graph. The notebook page is given below after running the cells.

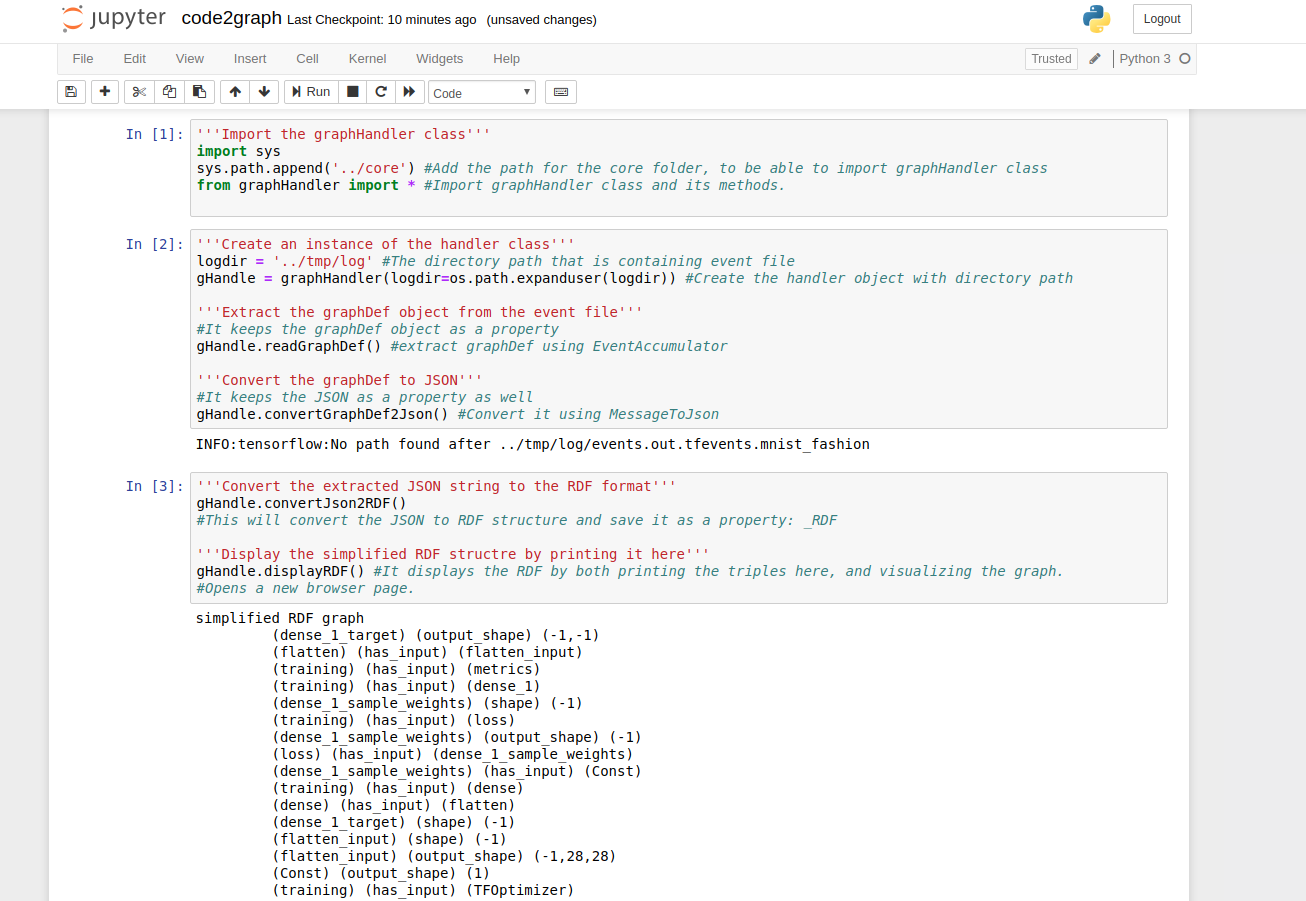


Figure 16: Test script walk through in Jupyter.

After running the last cell, an html page also pops up with an interactive RDF graph (drawn using pyvis opensource library [16]) viewer where all the components, the relation among the components can be viewed (see Figure 17).

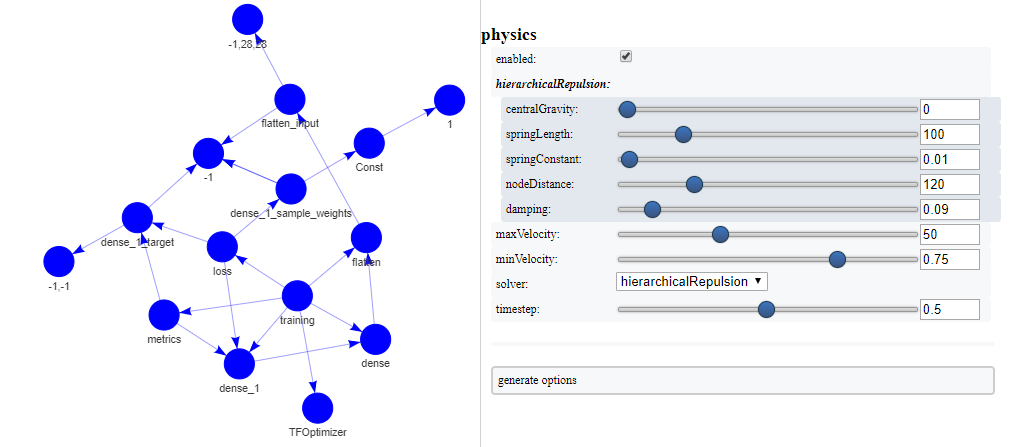


Figure 17: Plotting of simplified RDF graph.

# 8 Future Work

We have presented the updates to our work since Milestone 1. During this period, we worked on three aspects of graph generation: text2gaph, image2graph and code2graph. We implemented systems that extract scientific entities from publications (paperswithcode) using a BiLSTM and CRF models. We also described initial work on relationship extraction using a multi-task learning framework. We plan to extend these models by utilizing more contextual information such as trained scientific embeddings and ELMo embeddings to improve the quality of extracted relationships. We believe that this would lead to the creation of more accurate knowledge graph representations of scientific knowledge facts.

Regarding the image2graph component, we will improve the node detection algorithm for different types of DL diagrams. To improve text detection accuracy, we will incorporate a dictionary of possible DL layer names. This dictionary will be used to perform spell correction of the extracted OCR text. Then we will develop an algorithm to detect arrows present in DL diagrams. The direction of the arrows will give important information about the flow of the entire DL model design. Once nodes, arrows and text detection are done with reasonable accuracy, we will create the graph representing the flow information of the DL diagram.

Further work on code2graph will aim to develop more flexible stochastics methods for cleaning/simplifying the graphs extracted from code instead of the currently used manually created hard rules. We will also expand the present datasets with additional tensorflow/keras implementations, and will explore ways of automating or simplifying tasks of identifying code dependence on tensorflow or tensorflow with keras; and modification of code to supporting logging of the computational graph.

# 9 Potential Papers

We plan to submit our work to the WWW Workshop – [Knowledge graph technology and applications](https://datainnovation.soic.indiana.edu/www2019_kgta/index.html) and MOMACS. If reviews are favorable, we shall look to publish a full-length paper at EMNLP’19 and/or ISWC’19. [5]

# Works Cited

|  |  |
| --- | --- |
| [1] | [Online]. Available: https://paperswithcode.com/. |
| [2] | [Online]. Available: https://github.com/zziz/pwc. |
| [3] | "[Online] http://www.gitxiv.com/," [Online]. |
| [4] | [Online]. Available: https://www.w3.org/TR/shacl/. |
| [5] | L. Y. M. O. H. H. Yi Luan, "Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction," in *EMNLP*, 2018. |
| [6] | C. Clark and S. Divvalal, "PDFFigures 2.0: Mining figures from research papers.," in *IEEE/ACM Joint Conference on Digital Libraries*, 2016. |
| [7] | A. Sethi, A. Sankaran, N. Panwar, S. Khare and S. Mani, "DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers," *arXiv:1711.03543,* 2017. |
| [8] | K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition.," *arXiv:1409.1556,* 2014. |
| [9] | K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition.," in *IEEE conference on computer vision and pattern recognition*, 2016. |
| [10] | O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision,* vol. 115, no. 3, pp. 211-252, 2015. |
| [11] | [Online]. Available: https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/python/summary/writer/event\_file\_writer\_v2.py. |
| [12] | [Online]. Available: https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.json\_format-module. |
| [13] | [Online]. Available: https://www.tensorflow.org/versions/r1.13/api\_docs/python/tf. |
| [14] | [Online]. Available: https://rdflib.readthedocs.io/en/stable/. |
| [15] | [Online]. Available: https://www.tensorflow.org/tutorials/keras/basic\_classification. |
| [16] | [Online]. Available: https://pyvis.readthedocs.io/en/latest/index.html. |

1. [↑](#footnote-ref-2)
2. [↑](#footnote-ref-3)
3. https://euske.github.io/pdfminer/ [↑](#footnote-ref-4)
4. https://www.dbpedia-spotlight.org/ [↑](#footnote-ref-5)
5. https://scienceie.github.io/ [↑](#footnote-ref-6)