**DARPA ASKE DCC – Milestone 1, 2018**

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

Table of Contents

[1. Introduction 3](#_Toc531348122)

[2. Contributions 4](#_Toc531348123)

[3. Proposed Architecture 4](#_Toc531348124)

[4. Datasets 5](#_Toc531348125)

[5. Knowledge graphs – brief overview 6](#_Toc531348126)

[5.1 Data Models - RDF and RDFS 7](#_Toc531348127)

[5.2 Triple store and SPARQL 8](#_Toc531348128)

[6. Text2Graph 9](#_Toc531348129)

[6.1 Standard NLP pipeline 9](#_Toc531348130)

[6.2 Tools used: 11](#_Toc531348131)

[6.3 The proposed pipeline 12](#_Toc531348132)

[7. Image2Graph 13](#_Toc531348133)

[7.1 Extract figures from a DL paper 13](#_Toc531348134)

[7.2 Figure classification 14](#_Toc531348135)

[7.3 DL architecture diagram analysis 14](#_Toc531348136)

[7.3.1 Segmentation 15](#_Toc531348137)

[7.3.2 Text and symbol recognition 15](#_Toc531348138)

[7.3.3 Structural analysis 16](#_Toc531348139)

[7.3.4 Graph construction 16](#_Toc531348140)

[8. Code2Graph 16](#_Toc531348141)

[8.1 Generating RDF graph from the code 17](#_Toc531348142)

[9. Graph alignment: background 19](#_Toc531348143)

[9.1 Proposed graph alignment technique: 21](#_Toc531348144)

[10. Automatic code template generation 22](#_Toc531348145)

[11. References 24](#_Toc531348146)

# Introduction

This 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 #HR00111990010for the period from November 1st to November 30th 2018.

Under the description of Milestone 1 in the project plan of our proposal we have to submit a report describing details of the overall architecture, the algorithms and approaches that will be used during the implementation phase of the project. This report serves that purpose.

In today’s world, Artificial Intelligence (AI) techniques have become central to any data driven application. This growth has been spurned by the advances in the field of Deep Learning. In all Machine Learning (ML) conferences, the largest percentage of submitted and accepted papers deal with new DL models, architectures, or algorithmic approaches. Most of these papers are accompanied by source code which is publicly available. The growth is expected to increase in the future as DL keeps making major advances in medicine (e.g., by predicting the effect of mutations in non-coding DNA), and in Natural Language Processing (e.g., language translation). Keeping up with this large growth of DL publications and source code has been a challenge for the researchers and practitioners. Fortunately, the DL community has some established guidelines that try to manage this growth. More specifically the DL community has:

**Standardized frameworks:** Most DL frameworks are written on standard tools such as Tensorflow, Keras, PyTorch, etc. These frameworks are well documented and have an active community. Being able to rapidly prototype scientific results is a must for any community.

**Reproducibility:** Scientific communities such as physics and mathematics have always relied on the ability to reproduce results. The deep learning community that oversees modern conferences such as NIPS, ICML, ICLR, CVPR and KDD have emphasized the need for publications to accompany code that is reproducible and aligned with the results. This serves as a good benchmark for any scientific field.

**Open review system:**  Publication processes in the deep learning community also follow an open review system, thereby having a more thorough feedback of scientific advancements. This transparency in the field of scientific learning makes it ideal for machine curation and validation.

These standards seek to establish a common framework for the dissemination of scientific information. Our system (DCC) takes advantage of these processes and seeks to unify them by providing a *common machine queryable representation* of each scientific paper with all its modalities (text, image, code). This will allow scientists and engineers to explore the vast knowledge that resides in these papers and identify atomic scientific facts and their implementations. For example a scientist from a different field (e.g., psychology) interested in learning and implementing a Recurrent Neural Network (which is a specific DL model) can issue the following query (which currently cannot be done): “*Find all papers with a Bi-Directional Recurrent Neural Network model, implanted in Python and TensorFlow, in the area of Natural Language Processing, published between 2010 and 2018, in conferences like NIPS, ICML, ACL and EMNLP*.”

# Contributions

In this work we extract scientific facts from DL papers and represent them using RDF knowledge graphs (Consortium & others, 2014). RDF follows a W3C approved reference standard. It is also extensible to include domain representations utilizing ontologies. RDF allows for easy and shallow representation of scientific facts as a <subject, predicate, object> or SPO triple statements. A scientific publication consists of facts in various forms, such as text, images, tables and code. Some of the information is shared across representations. Given that most scientific advancements require the conceptualization, explanations and reproducible implementations, it is important to have a uniform representation that holistically represents a scientific publication (text, images code, etc.) as a machine curatable model. This allows for easy dissemination of scientific facts. Thus, the contributions of our proposal can be summarized as follows:

* A deep knowledge graph (DKG) that captures the essence of a scientific publication (images, text and code) in a standardized format
* Automatically extracting atomic scientific facts in various modalities to the same standard format.
* Expressive querying over a multitude of scientific facts, rather than just textual querying.
* Quantify scientific novelty by comparing new and existing publications.
* Ability to suggest automatic implementation code templates for new and existing publications.

# Proposed Architecture

The overall architecture of our approach is shown in Figure 1, which consists of two main parts. The first part shows our curation architecture for the system. Scientific publications, along with their source codes are first sent to a graph extraction module. This module extracts atomic facts from text, images and code. These facts are diverse and contain overlapping and complementary information. To convert this to a standard KG representation model, they are sent to a mapping system, that converts these facts to RDF triples with the right vocabularies. In addition to this, the mapping system also maps them to their representative domains specified by ontologies (e.g., CNN is a concept in the DL domain, which is a sub domain of ML). After this is completed, the knowledge graphs are aligned based on their structural similarity. This final aligned knowledge graph is then stored in a data store known as triple stores. These triple stores can then be queried by the rich pattern query language SPARQL, that would allow expressive querying with complex conditions. For example, a complex query such as “Find all AttentionModels and their implementations in DL papers published in NIPS and ACL” can be transformed easily to SPARQL.

In the second component (at the bottom of the figure), we show how the curated KG can be used to automatically infer code for papers that do not have implementation details. First an aligned knowledge graph without source code is extracted from the paper. The corresponding code graph is then inferred by using previous related papers that have source code.

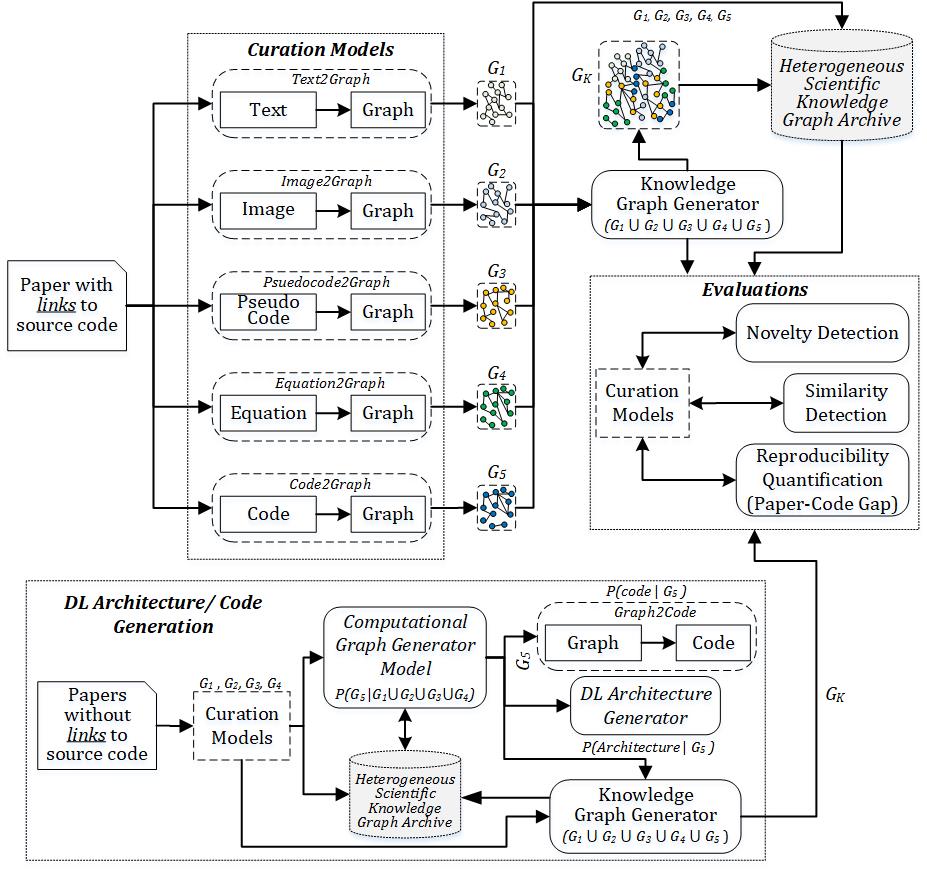


Figure : General pipeline for DCC

In the following sections of this report we provide details on the major components of the pipeline shown in Figure 1. We first start with a discussion on the datasets that are required for extraction, training and validation. Then we proceed to briefly discuss the knowledge graph representation model followed by text2graph, image2graph and code2graph. Each of these sections also discuss related work and state-of-the-art tools inline.

# Datasets

We have collected a large number of scientific papers, in PDF format, that describe a wide variety of deep learning models, architectures, and applications. We have developed a Python web scraper that can fetch the latest papers from the following sites:

1. Papers with Code[[1]](#footnote-2) – This is a searchable site that links machine learning papers on ArXiv with code on GitHub. The site also tags any framework libraries used along with other information such as stars on GitHub. 100 papers have been initially downloaded and stored in a local hard drive. The size of this dataset is 1,7GB.
2. PWC[[2]](#footnote-3)– This is a GitHub repository consisting of a CSV with 7,737 papers with code. Unlike the previous site, this list contains many broken links to both PDFs and source code. We developed a web scraper in Python to fetch the papers that have a valid ArXiv PDF link and that have a valid github source code repository. This dataset is 830Mb and consists of 69 papers. The web scraper can be extended to papers hosted outside of ArXiv and source code hosted outside of Github.
3. GitXiv[[3]](#footnote-4)– It is a curated feed of computer science projects. Each project is presented as a link to the project's arXiv repository, the link to the Github repository, other links, and a discussion. Although we have not written the web scraper to download the papers, source code, and available meta-data, we list this data source here for documentation purposes

# Knowledge graphs – brief overview

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. This is the foundation for the Resource Description Framework (RDF), which is the keystone for the Semantic Web. This structured machine accessible format has enabled large scale adoption of RDF graphs for a variety of AI applications. The data that is expressed by these graphs can be broadly classified into the following two types:

**Schema data (TBox):** Domain specific knowledge encoded by means of class and property hierarchies. In addition, this also includes domain and range definitions. Schema data represents the conceptualizations of a set of facts. The schema data is generally static and once created, doesn't change frequently.

**Instance data (ABox):** Individual instances of the classes and statements describing the instances. Instance data adds more facts to the existing conceptualizations. In the RDF context, instance data associates entities with their predicate and object values. Since most knowledge graphs are created automatically, instance data population is dynamic and keeps changing at regular intervals.

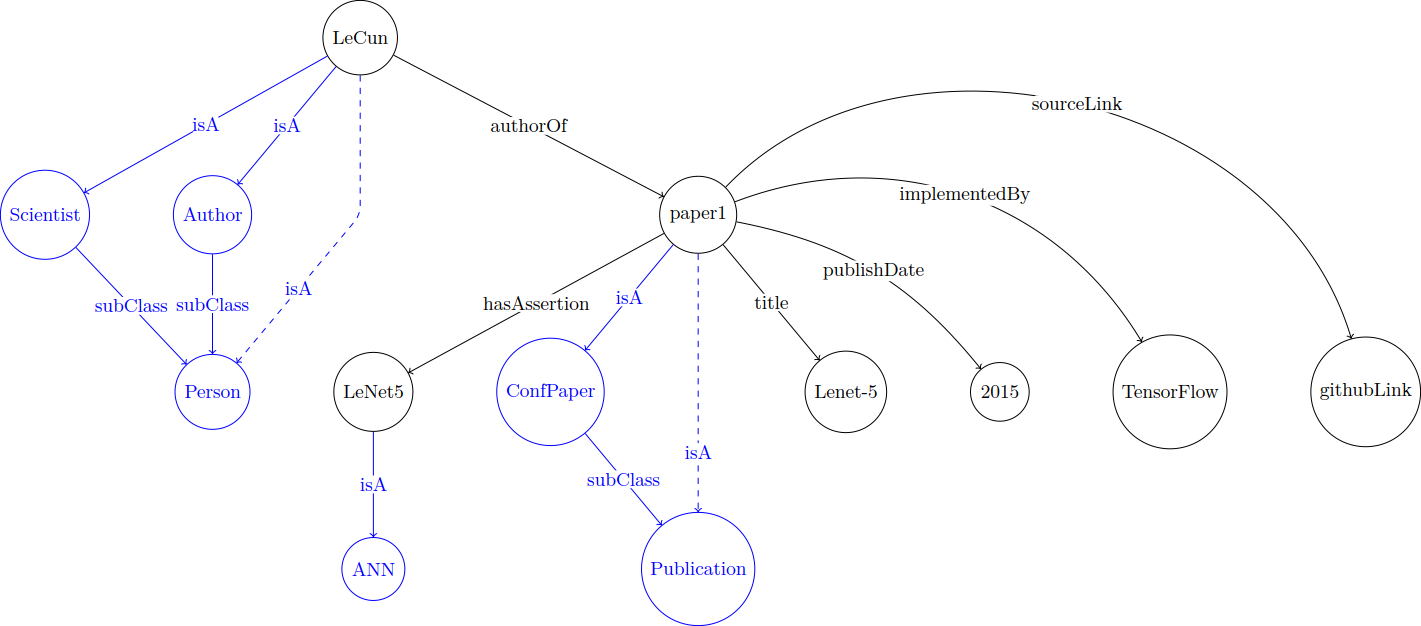


Figure : Proposed KG snippet of LeNet-5

Representing data as schema and instances allows a great degree of expressivity in the data. Thus, popular knowledge graphs like YAGO (Suchanek, et al., 2008), DBpedia (Auer, et al., 2007) and Freebase (Bollacker, et al., 2008) utilize RDF and RDFS to encode information in the billions. Extending this to a scientific publication leads to Figure 2, which shows a representative RDF knowledge graph of the LeNet-5[[4]](#footnote-5) paper. The instance data is shown in black and schema data in blue. Instances have properties, for example, the instance LeCun has the properties authorOf and the instance paper1 has properties title, publishDate, implementedBy, sourceLink. The conceptualizations or the domain knowledge are the concepts *Scientist, Author, Person, ConfPaper, Publication.* Even from this representative example, it is evident that RDF serves as an expressive representation format which can be queried effectively for facts that were hitherto impossible in normal search systems. In addition to this the dashed arrows represent *inferred relations.* (Eg: LeCun is *inferred* to be a Person, paper1 is *inferred* to be a Publication).

## 5.1 Data Models - RDF and RDFS

The Resource Description Framework (RDF) is a formalism to describe web resources. In RDF, a resource is always referenced by a Uniform Resource Identifier. This resource is connected to other resources by means of properties. Thus, properties in RDF parlance are the edges in the Knowledge graph terminology. The properties can point to two different kinds of values. If they point towards resources, they are called object type properties and if they point towards literals they are known as datatype properties. Thus, datatype properties describe the attributes of a property itself and object type properties relate a resource to other resources. RDFS is an extension of RDF to include domain specifications (Brickley, Guha, & McBride, 2014). This is done through class definitions and hierarchies for properties and classes. In addition, there are also restrictions on the kinds of values that classes and properties can be part of. This is done by domain and range restrictions.

The resources of RDF can be divided into groups called classes. The members of the class are known as instances of the class. An instance of a class uses the type property to associate itself with a class. An instance can be associated to more than one class as well. Classes by themselves can also be grouped under other classes, thus forming a class hierarchy. This is specified by the subClassOf property. Predicates also can be part of a hierarchy and they are specified by the subPropertyOf property. In addition to this the properties domain, range define the domain and range of the property values. For our proposal we utilize the five property statements type, subClassOf, subPropertyOf, range and domain

## 5.2 Triple store and SPARQL

RDF and RDFS facts are generally stored SPO triple statements in triple stores. The knowledge graph shown in Figure 2 will be stored as the table Table 1 below. These triples stores are queried by a pattern query language called SPARQL.

|  |  |  |
| --- | --- | --- |
| Subject | Predicate | Object |
| Scientist | **isA** | **Person** |
| Author | **isA** | **Person** |
| ConfPaper | **isA** | **Publication** |
| LeCun | **isA** | Scientist |
| LeCun | **isA** | Author |
| LeCun | **authorOf** | paper1 |
| paper1 | **Title** | LeNet-5 |
| paper1 | **publicationDate** | 2015 |
| paper1 | **implementedBy** | TensorFlow |
| paper1 | **sourceLink** | githubLink |

**Table 1:** Table view of KG from Figure 2

Querying RDF triple stores is done by the popular graph query language SPARQL. SPARQL is a graph-matching pattern query language. SPARQL queries are query patterns used to query RDF triple stores. It consists of two parts

**Pattern matching part:** This consists of a set of triple patterns, along with other operators like OPTIONAL,UNION. Triple patterns are just triple statements replaced with variables.

**Solution modifiers:** This part consists of applying operators like SELECT, DISTINCT, ORDER, LIMIT on the evaluated values of the query pattern.

A complex query such as “Select authors, publications with a title of LeNet-5 which were published in 2015” can be composed with three pattern statements. This is shown below as:

|  |
| --- |
| SELECT ?author ?title WHERE {  ?author authorOf ?pub .  ?pub isA Publication .  ?pub title LeNet-5  ?pub publishDate “2015”.  } |

This query, when run on Table 1 will give the results *LeCun, pub1.* Similarly on larger triple stores, more complex and expressive pattern queries can be constructed.

In the next few sections we detail the components that will populate our knowledge graph

# Text2Graph

Scientific publications, unlike news articles, have both common internal structure and structured relations to the world. By internal structure we mean that these publications usually have information on authors, related works, descriptions of models and/or methods and how they relate to prior art, experimental structure and experimental data. In essence, each paper can be represented by a knowledge graph, but the entities and relations in it overlap or link to the overall ontology, thereby expanding the ontology and placing the paper in the context of other work. From each article we extract entities and relationships using Natural Language Processing (NLP) methods and construct entity-relation triples. We will then connect them to existing knowledge graph entries (e.g. from DBPedia or from our own ontology).

Fortunately, the advances in NLP technique over the recent years enable us to leverage effective state-of-the-art tools for many steps of this task. One example framework for text2graph is described in (Kertkeidkachorn & Ichise, 2017). Below we describe at a high level the standard NLP pipeline. We then give brief descriptions of the tools we will leverage, and the specify the text2graph pipeline we will build.

## 6.1 Standard NLP pipeline

Understanding natural language involves multiple steps, the most common of which are:

1. *Sentence/word tokenization:* the process of breaking a stream of textual content up into sentences, words, terms, symbols, or some other meaningful elements called tokens. A token can be also defined as an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing.
2. *Text stemming and lemmatization:* reducing the inflectional forms of a word to a common base form. Stemming refers to the process of reducing inflection in words to their root forms, even if the stem is not a valid word. On the other hand, lemmatization reduces the inflected words ensuring that the root is a valid word.
3. *Parts of speech (POS) tagging:* assigning to each word POS labels such as noun, verb, adjective, etc. Knowing the role of every word in a sentence helps to determine the meaning of the sentence. There are mainly two categories of POS tagging approaches: (a) rule-based and (b) machine learning based (e.g., HMM, and DNN).
4. *Dependency parsing:* building a ‘parse’ tree that establishes relationships between parent words and words which modify them. The most recent parsers implement deep neural network technologies accepting as input word embeddings (Chen D., 2014).
5. *Named entity recognition (NER):* identifying the names of entities in a sentence. Named entities can be generic proper nouns that refer to locations, people, or organizations. They can also be much more domain-specific, for example in a biomedical context they can refer to diseases or genes. In our project they could be names of algorithms (ex. recurrent neural network) or particular architectures such as LeNet5. Early NER methods used hand-crafted features and supervised methods in the form of rule-based systems and sequence labeling algorithms such as HMM, SVM and Conditional Random Fields (CRF). Modern NER methods increasingly use neural architectures such as the one proposed by (Lample, et al., 2016), where a combination of a bidirectional LSTM and a CRF are used.
6. *Relation extraction (RE):* detecting and classifying predefined relationships between entities from the text. RE can transform unstructured text into more structured form which then can be used to perform tasks such as knowledge base population and question answering. For example, one common relation is born-in, which holds between a person and a geographical location. Relations are extracted by analyzing the sentences using POS taggers, dependency parser, and named entity recognizer. RE models based on deep learning architectures have been recently proposed which demonstrate better performance than traditional methods for RE which depend on hand-crafted features. In (Li, et al., 2016) an unsupervised pre-training method is developed. The sequence-to-sequence model for deep RE is used and the pre-trained models need only half or even less training data to achieve equivalent performance as the same models without pre-training. In (Shen & Huang, 2016) a novel attention-based convolutional neural network is developed for RE. The model makes full use of word, POS tags and position embeddings to determine which parts of a sentence are more influential. The reported experiments show that the model achieves better performance than other DNN models by using minimal feature engineering. In (Santos, et al., 2015) a CNN is used to perform classification ranking of relations in text. The method uses a novel pairwise ranking loss function that makes it easy to reduce artificial classes and helps it achieve better results than state-of-the-art methods. A recent review of the latest trends in deep learning based NLP can be found in (Young, et al., 2018), where the authors review a large number of models and methods.

We should note that state-of-the-art NLP tools, which we will discuss further, provide models and implementations of methods for performing many of these tasks, at least for general English texts.

## 6.2 Tools used:

In this section we briefly review the main NLP libraries we will consider in our project for building the above mentioned NLP pipeline.

* **SpaCy[[5]](#footnote-6)** is a relatively new framework and has gain popularity in the Python NLP community. It is very fast as it is written in Cython which is a superset of Python that compiles to C yielding remarkable performance boosts. It supports POS tagging, Named Entity Recognition, and Dependency Parsing (for which it is the fastest and one of the best-performing tools (Choi, et al., 2015)). SpaCy comes with its own deep learning framework (Thinc) that is tailored and optimized for NLP tasks. It can also easily incorporate TensorFlow, Keras, and Scikit-Learn allowing NLP researchers to develop their own machine/deep learning algorithms.
* **Stanford CoreNLP[[6]](#footnote-7)** is a comprehensive, fast and modern grammatical analysis tool. It is written in Java which allows it to be able to solve complex NLP tasks in a very efficient way. It provides a number of APIs for many other programming languages, including Python. Features that make CoreNLP attractive are its POS tagger, named entity recognition, parser, coreference resolution, bootstrapped pattern learning and open information extraction tools. In addition, an annotator pipeline can easily include additional custom or third-party annotators. CoreNLP is designed to be flexible and extensible, and it can provide a great foundation for building microservices because it can run as a web service.
* **NLTK[[7]](#footnote-8)** is perhaps the most well-known NLP library. It is written in Python and has a big community behind it. It has tools for almost all NLP tasks and is widely used for tokenization, lemmatization, stemming, parsing and POS tagging. Although it is relatively fast library, it is slower than SpaCy and does not support deep neural network models. However, it has APIs integrating SpaCY and CoreNLP.
* **Gensim[[8]](#footnote-9)** is an NLP library, written in Python, specializing in ‘topic modeling for humans’, which is a technique for extracting the underlying topics from large volumes of text. It uses LDA to create high-quality topic models. It is a convenient package for processing text, using word vector models (e.g., Word2Vec or FastText). An advantage of Gensim is that it can handle large text without having to load the entire file into memory.
* **DBpedia[[9]](#footnote-10)** is a community project that extracts knowledge from Wikipedia and makes it available via established Semantic Web standards and Linked Data best practices. Wikipedia itself can only offer very limited query and search capabilities, due to the lack of the exploitation of the inherent structure of its articles. One of the main goals of DBpedia is to provide comprehensive querying and searching capabilities to a wide community by extracting structured data from Wikipedia, which can then be used for answering expressive queries.
* **DBpedia Spotlight[[10]](#footnote-11)** is a tool that can automatically annotate mentions of DBpedia resources in text. DBpedia Spotlight is capable of recognizing that entities have been mentioned and subsequently matches these names to unique identifiers (e.g., “dbpedia:Recurrent neural networks”). It has been used for building solutions for NER, key-phrase extraction, tagging, amongst other IE tasks. The tool is implemented in Java; and can be deployed as a server.

**Snorkel[[11]](#footnote-12)** is a system for rapidly creating, modeling, and managing training data. It been used for building relation extraction models from weakly supervised data (Ratner, et al., 2018).

## 6.3 The proposed pipeline

The objective of our text2graph pipeline, shown in Figure 3, is to extract relevant facts from text derived from scientific publications and convert it to a knowledge graph.

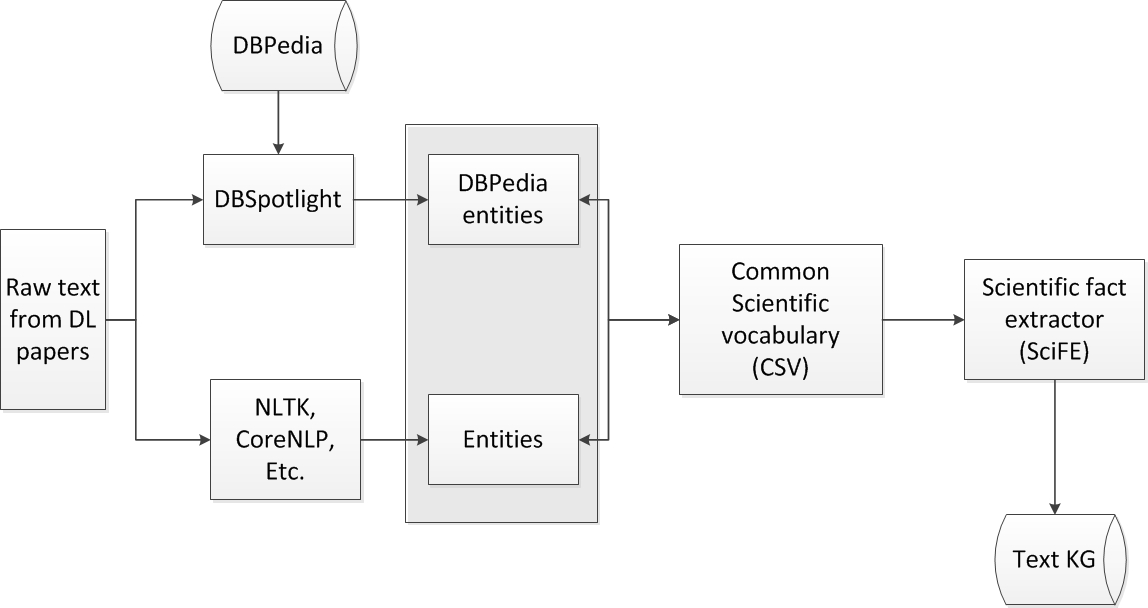


Figure 3: Text2Graph Pipeline

The raw text can be extracted from the publication in a multitude of ways, for example with PyPDF2[[12]](#footnote-13) module. We note here that the raw text can be the entire pdf paper or some sections of it (e.g., abstract, introduction, proposed method, or conclusions). Once we have the raw text, the first step is to identify the relevant entities. We leverage DBpedia Spotlight to get an initial set of uniquely identified entities corresponding to DBpedia entries. This initial set is likely to contain some errors due to possible multiple interpretations of terms. For example CNN will map to the international news channel, rather than to the Convolutional Neural Network architecture. We will perform manual review of the extracted mappings. More significantly, the mapping will be very much incomplete, since DBPedia does not contain less common technical terms that abound in Deep Learning literature. Therefore, we will use the entities mapped to DBPedia as seeds for a training set for an automated pipeline to extract a wider set of entities using NLP tools such as those discussed above. Once we develop a Common Scientific Vocabulary (CSV) for Deep Learning, we utilize it to build Scientific Fact Extractor (SciFE) models for identifying relations between entities of interest. This will result in a collection of facts, represented as triplets, that will serve as a core for the graph representing scientific textual information. These triplets will be mapped to their appropriate RDF graph triple representations.

From the extensive set of tools, we chose SpaCy for the initial implementation due to the quality of its results, speed of execution and ease of getting implementing pipelines. Other NER tools require extensive annotations (Ratinov & Roth, 2009), while Spacy requires just positional information. These features will enable rapid prototyping for DCC.

# Image2Graph

In this section, 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 1: (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 this work, we also introduce a fully-annotated real-world dataset to demonstrate the efficacy of our parsing approach.

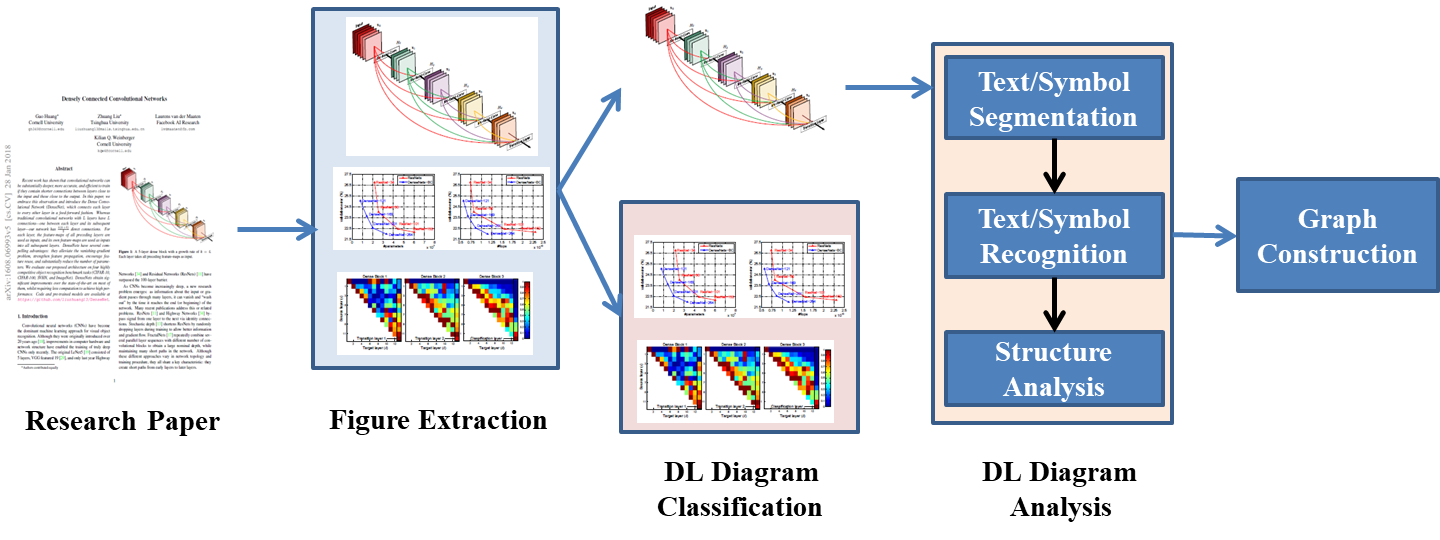
****

Figure 4: Architecture of the Image2Graph module

## 7.1 Extract figures from a DL paper

Extensive research has been done on extracting of visual figures from a PDF document (Choudhury & Giles, 2015). It is a challenging task to extract a figure as a whole as the vector images are generally embedded in the PDF document and a large figure may have multiple sub-figures. To this end, we plan to use publicly existing tool named PDFFigures 2.04 (Clark & Divvalal, 2016) for extracting a list of figures from a research paper.

## 7.2 Figure 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 propose to use 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 plan to employ three network architectures: AlexNet (Krizhevsky, et al., 2012), VGG19 (Simonyan & Zisserman, 2014) and ResNet-50 (He, et al., 2016). All the three networks are pre-trained on the 1.2 million images from ImageNet (Russakovsky, et al., 2015) and then will be fine-tuned for our figure classification task.

It has been observed that the DL architecture diagrams typically do not follow any definition and show extreme variations. Sethi et al. (Sethi, et al., 2017) divided the model diagrams into 5 broad categories. The five types are: (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.

Once the DL figures are identified automatically, we plan to employ another 5 class neural network classifier to identify the category of the input DL architecture figure. These two steps can be combined into one by training a 6 class neural network classifier. We plan to explore both options and the one giving higher accuracy will be chosen.

For phase 1, we plan to consider only the box-based neural network architectures (2D Box, stacked2D Box and 3D Box) for further analysis. Neurons plot and Pipeline plots will be considered in the second phase.

## 7.3 DL architecture diagram analysis

Diagram recognition is a well-studied problem (Blostein, 1995). However, most of the works perform analysis on flowcharts, UML diagrams, finite automata (Bresler, et al., 2016). (Wang, et al., 2017), which follow predefined structure. Unlike these diagrams, DL architectures do not follow any specific structure and formatting, thus making the analysis very challenging. To the best of our knowledge, this is the second effort to automatically analyze DL diagrams after previous work by (Sethi, et al., 2017).

Diagram analysis task can be divided into three steps, (i) Segmentation to isolate symbols, (ii) Symbol recognition, and (iii) Structural analysis for identification of spatial and logical relationships among symbols. We will follow the same process in DL diagram recognition in a broad way; however, the properties differ. In general, types of symbols (e.g., rectangle, diamond, circle, oval, parallelogram) representing different operations in a flowchart is not common in DL diagrams. Mostly, different colored/sized rectangles with text descriptions are used to represent node or layer information. Arrows in a DL flowchart can have various shapes, and can point to almost everywhere. All these add difficulties in DL diagram recognition.

### 7.3.1 Segmentation

Segmentation of a diagram into text, symbols and interconnections is an important stage, as it directly affects interpretation of the diagram. If a document image consists of multiple diagrams then these diagrams need to be segmented as well. The segmentation between diagrams can be done based on the projection profile (Pavlidis & Zhou, 1991) as it shows a definite gap between the two diagrams.

Graphical symbol (arrows, boxes, glyphs) segmentation is performed in order to isolate the individual symbols (Herbrich, et al., 1999) of the diagram including connectors (usually lines/arrows). Segmentation methods (Pujol, 2002) are generally of three types: (i) based on global knowledge about an image, (ii) edge-based segmentation, and (iii) region-based segmentation. As the image is binarized, the image consists of contrasted objects located on a uniform background. One possible approach is to automatically detect the objects/symbols (Tirkaz, et al., 2012) (Guo, et al., 2012) (Santosh, et al., 2012) 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) (Li, et al., 2018) framework, regularization on the attention maps will be used to guide the learning of detecting the DL building blocks more efficiently and effectively.

### 7.3.2 Text and symbol recognition

Once the segmentation is done, symbol recognition will be performed. We treat arrows in a special way since their form and shape vary, which is a difficulty for traditional classifiers. There are two stages. First, uniform symbols are recognized using a standard classifier. Second, arrows are detected as connectors between symbol candidates found earlier.

The problem of text recognition is straightforward with many existing solutions to recognize words, phrases, math equations and text blocks. We plan to use Tesseract[[13]](#footnote-14) to recognize the text in each node/ layer. A dictionary of possible DL layer names will be created to perform spell correction of the extracted OCR text. According to (Sethi, et al., 2017), a layer description is generally provided within a detected block or in its vicinity.

### 7.3.3 Structural analysis

Structural analysis is used to find spatial and logical relations among the symbol/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 plan to employ a combination of logical structure of a DL diagram with graph grammar to perform the flow detection. We will create a grammar describing valid deep learning design models defining the list of possible next layers for a given current layer. The arrow directions along with possible valid next node/layer information will be used to generate valid flow detection.

### 7.3.4 Graph construction

After detecting DL design flow, a RDF graph will be created to represent the extracted information like the one shown in Figure 2(a) which is an image of the AttentionModel taken from (Hu, et al., 2018).

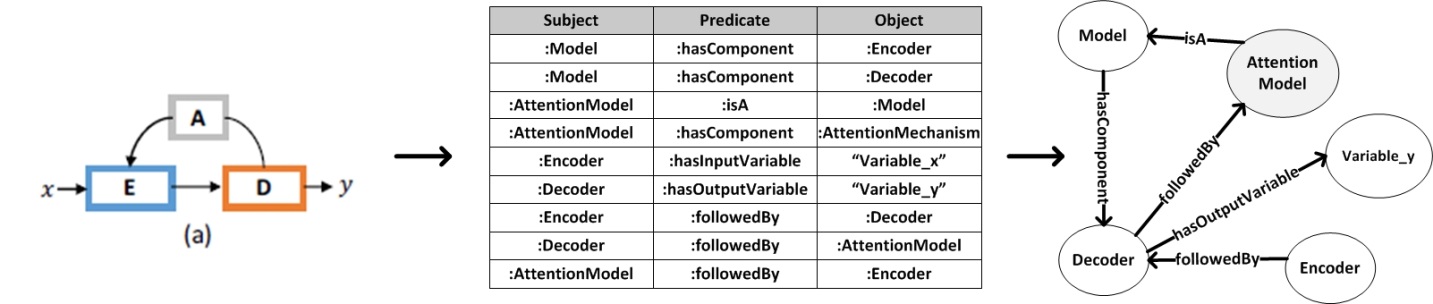


Figure 5: Information extraction from images and conversion to KG

# Code2Graph

The dataset utilized in this project will consists of scientific papers that utilize/propose/extend the deep learning architectures with their corresponding codes written in python using TensorFlow. The aim of this segment of the project is to convert the corresponding codes that implement the deep learning architectures written in python to a knowledge graph. These scientific papers 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 by default converts all the user program written in the python that implements the scientific papers into the dataflow graph. These dataflow graph consists of nodes represented by the computation. Moreover, the edges also show control dependencies among various computation nodes. At the fundamental level, every operation is represented as a node and every Tensor is represented as an edge, coming in or out of the nodes, in the dataflow graph. As an example, let’s take the following operation:

1. **import** tensorflow as tf
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)

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

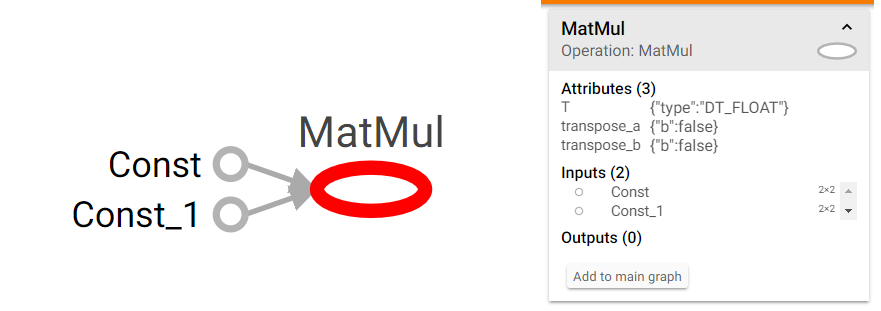


Figure 6: TensorFlow graph visualized using TensorBoard

However, the codes written in python that implement in scientific paper, however, 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 which are used in most of the knowledge graph. This knowledge graph will be used to create the super graph later.

## 8.1 Generating RDF graph from the code

The pipeline for converting the scientific code the RDF graph is shown in Figure 5.



Figure 7: Pipeline for code to RDF graph

The details of each of the steps are described as follows:

1. **Pre-processing the dataset code repositories:** The first steps in the code2graph pipeline consists of manual and automatic pre-processing to make sure all the dependent functionalities are implemented in TensorFlow. In this step we remove the dependency of the code on libraries, functions, etc., that may not be converted to the TensorFlow computation graph. Rather, these portions of the code are filtered and explicitly converted to TensorFlow graph using AutoGraph functionality. It converts the python code into equivalent TensorFlow Graph. In the pre-processing stage, we will also detect various variables, functions, modules, etc., to check if they have proper and standard name scope which will be used as node or entity names later while constructing the graph.
2. **Recreating the environment for the codes:**The scientific codes will have been implemented using various version of libraries.To extract the computational graph, the first step will be making sure that the libraries and dependencies necessary for running the scientific codes are available for compiling the dataflow graph and extract the information. If possible, all the scientific codes will be compiled in a single environment (latest Tensorflow, Python version, etc.) to have a consistent graph representation of the deep learning architectures implemented in the scientific papers.
3. 

Figure 8: Example of TensorFlow dataflow graph

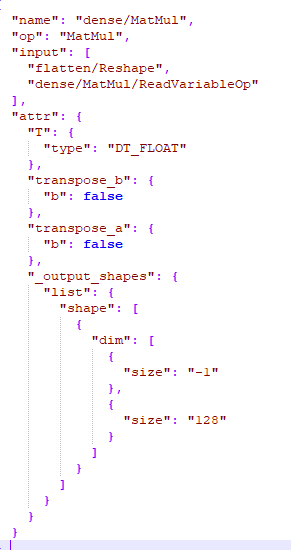
1. **Extracting the tf.Graph():**When the scientific codes are written in client TensorFlow program such as python, TensorFlow creates an instance of the Class **tf.Graph().** This graph represent the TensorFlow computations represented as a dataflow graph. It contains the nodes represented as the operations and the data (Tensors). This dataflow graph can represent the deep learning architectures at various level of resolutions (shown in Figure 6). This dataflow graph with proper context information can be passed to the TensorFlow master core runtime to run and test the deep learning architecture. Moreover, this graph can be parsed to acquire the underlying user code that was used to generate the graph in the first place.
2. **Extracting the tf.GraphDef()*:*** The tf.Graph() is converted to the serialized structure data format using tf.GraphDef() protocol buffer. This data structure consists all the necessary information to describe the dataflow graph. Hence, we can extract this data structure to convert it into the RDF data structure. However, we will first store all the tf.GraphDef() of the dataset using tf.summary.FileWriter().

Figure 9: JSON format of the tf.GraphDef()

1. **Converting tf.GraphDef() to Json:**After acquiring the tf.GraphDef() protocol buffer description of the tf.Graph(), it is converted to a JavaScript Object Notation (JSON) format. The JSON sample of the MatMul operation inside the Dense operation is shown inFigure 7***.***
2. **Converting Json to RDF graph:**After acquiring the JSON format data, we will first filter it to separate the attribute of the node, contexts necessary for generating the tf.Graph(), and create a RDF triples consisting of subject, predicate, and the object <s,p,o>. The RDF graph parser will be able to generate different resolution of RDF graph. We will generate RDF graph for higher level model (such as the sequential model used in TensorFlow Keras), lower level APIs (with detailed computational graph), etc. Each of these graphs will aid in constructing super graph or during code inference. The higher-level APIs based RDF graph will be easier to align with the deep learning architecture graphs curated from text, images, etc. Whereas, the lower level APIs carry rich information necessary for curating information later from the super graph about the most likely attributes of the operations (such as datatype, shape, etc.,), which will help in filling out the missing out information after the code RDF has been inferred from the super graph. We will utilize the python library rdflib to create the python RDF graph. The sample RDF is shown as follows:

<**model, hasInput, flatten\_input>**

**<model, hasLayer, Dense>**

**<model, hasLayer, Dense\_1>**

**<model, hasTrainer, TFOptimizer>**

1. **Evaluation of code2graph:**To evaluate the performance of code2graph, we will require a feedback from graph2code pipeline to check if the generated graph fully comprises all the necessary information to re-construct the computational graph and re-produce the original results produced by the testing code from the scientific code dataset. Hence, we will merge the evaluation of code2graph and graph2code together.

# Graph alignment: background

Traditionally, knowledge graph alignment has always referred to the process of ontology alignment. An ontology is a formal representation of some world and domain knowledge. In this work, we utilize a shallow ontology to define concepts that underlie the creation, assimilation and dissemination of scientific publication documents. This was shown in the example in Figure 2 . This ontology was also populated by *instances* or *entities* that are instantiations of the abstract concepts. Thus, knowledge graph alignment owes its foundations to the early work done in ontology matching. Given heterogeneous ontologies, each describing the same concept or part of the same concept, it became clear that aligning them under a unified schema would enable better *interoperability*. Despite these achievements, large scale ontology alignment has always remained a challenge. Being able to resolve definitions of varying complexities remains a key problem. Concepts can be defined in many possible ways and no single metric considers all the structural similarities and differences required for alignment. In our work, we use knowledge graphs as shallow representations of ontologies. All our extracted facts populate the ontology as instances or entities of the domain. This gives us the scope and the flexibility of using lightweight semantics and still rapidly construct a graph to represent a scientific paper. Given this, our alignment technique will involve both alignment at the conceptual level and alignment at the instance level. Based on this, we look at some of the different approaches used for knowledge graph alignment and then propose an initial direction for our work.

Graph alignment approaches from the logic world can be classified under two broad schemes. The first kind utilizes the structure and semantics of the underlying graph to suggest nodes that can be aligned. In the second kind, structural properties of the instances are used to align the graph. These approaches are briefly summarized below:

**Ontology based approaches:** In ontology-based approaches, the semantics of OWL properties and class definitions are utilized as similarity metrics to determine the ability to map concepts. Traditional approaches to ontology matching have focused on either aligning the classes or aligning the instances. The approaches that align classes include techniques such as sense clustering (Gracia, d'Aquin, & Mena, 2009), lexical and structural characteristics (Jean-Mary, Shironoshita, & Kabuka, 2009) and composite approaches (Aumueller, Do, Massmann, & Rahm, 2005). These approaches do not consider sub classes or sub properties in an ontology and thus don’t scale well to modern knowledge graphs, which also encode taxonomies or schemas. **Instance based approaches:** In instance-based approaches, techniques generally use the terminological structure [4], logical deduction [5] or cluster by relations (Bhattacharya & Getoor, 2007). As is the case with ontology-based approaches, instance based approaches utilize only part of the knowledge graph.



Figure : Knowledge graph alignment - text, image, code

## 9.1 Proposed graph alignment technique:

In our knowledge graph, we encode both domain concepts(ontologies) and instances of these concepts. So our alignment vision follows an approach that utilizes the structure of both the ontology data and instance data to look for nodes that require alignment and linking.

This is shown in Figure 10, where we refer to three diverse knowledge graphs, each representing different modalities of a scientific publication. Each of these knowledge graphs encodes hierarchies and taxonomic concepts as schema definitions and scientific facts as instances of these concepts. To align this, we propose a technique that utilizes:

**Concept Alignment:** Utilizing structural features we propose that we align concepts that have similar definitions across schemas.

**Instance Alignment:** An instance data aware methodology, where the features of the nodes are considered to define a similarity metric. This metric is then used to compute similarities between candidate nodes. Define a threshold metric to merge nodes features that are common above a threshold value. Using this we link the text2graph, image2graph and code2graph. to form a unified domain knowledge graph. We also propose to utilize the similarity metric to augment existing state-of-the-art entity alignment techniques for knowledge graphs and merge entity nodes.

The resulting merged union graph is also shown at the bottom of Figure 10.

# Automatic code template generation

After the super-graph is created, the RDF graph for the code can be inferred. 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. The pipeline for converting inferred RDF graph to the code is shown in Figure 8, and each of the steps is explained as follows:



Figure 11: Pipeline for converting RDF graph to code

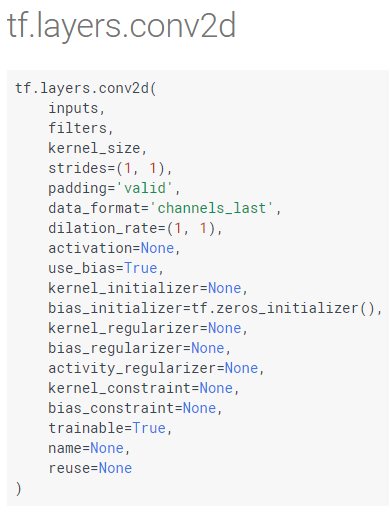
* 1. **Rule-based preprocessing of the RDF graph:**The inferred RDF graph is probabilistic, which means it cannot be directly converted to the computational graph as some of the suggested nodes in the graph may not be possible to implement in the TensorFlow using the dataflow graph. Moreover, the suggested 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.
  2. **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 9, 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, we when the super-graph is aligned and created, we will query the super graph to acquire additional possible attributes if available. Moreover, some of these attributes may 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.).

Figure 12: 2D convolutional lower level API arguments

* 1. **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.
  2. **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.
  3. **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.

# References

Blostein, D., 1995. *General diagram-recognition methodologies..* s.l., s.n., pp. 106-122..

Bresler, M., Průša, D. & Hlaváč, V., 2016. Online recognition of sketched arrow-connected diagrams.. *International Journal on Document Analysis and Recognition (IJDAR),* pp. 253-267.

Chen D., M. C., 2014. *A Fast and Accurate Dependency Parser Using Neural Networks.* s.l., In Proc. EMNLP.

Choi, J. D., Tetreault, J. & Stent, A., 2015. *It Depends: Dependency Parser Comparison Using A Web-based Evaluation Tool..* s.l., s.n.

Choudhury, S. R. & Giles, C. L., 2015. *An architecture for information extraction from figures in digital libraries..* s.l., s.n., pp. 667-672.

Clark, C. & Divvalal, S., 2016. *PDFFigures 2.0: Mining figures from research papers..* s.l., s.n., pp. 143-152.

Guo, T., Zhang, H. & Wen, Y., 2012. An improved example driven symbol recognition approach in engineering drawings. *Computer & Graphics ,* pp. 835-845.

He, K., Zhang, X., Ren, S. & Sun, J., 2016. *Deep residual learning for image recognition..* s.l., s.n., pp. 770-778.

Herbrich, R., Graepel, T. & Campbell, C., 1999. *Bayes point machine: Estimating the Bayes point in kernel space.* s.l., s.n., pp. 23-27.

Hu, Z. et al., 2018. Texar: A Modularized, Versatile, and Extensible Toolkit for Text Generation. *arXiv:1809.00794*.

Kertkeidkachorn, N. & Ichise, R., 2017. *T2KG: An End-to-End System for Creating Knowledge Graph from Unstructured Text.* s.l., s.n.

Krizhevsky, A., Sutskever, I. & Hinton, G. E., 2012. *Imagenet classification with deep convolutional neural networks.* s.l., s.n., pp. 1097-1105.

Lample, G. et al., 2016. *Neural Architectures for Named Entity Recognition,* San Diego, CA: Proceedings of NAACL-HLT.

Li, K. et al., 2018. Tell me where to look: Guided attention inference network. *arXiv:1802.10171*.

Li, Z., Qu, L., Xu, Q. & Johnson, M., 2016. *Unsupervised pre-training with Seq2Seq reconstruction loss for deep relation extraction models.* Brisbane , Australia, Proc. Australian Language Technology Association Workshop.

Pavlidis, T. & Zhou, J., 1991. *Page Segmentation by White Streams.* s.l., s.n., pp. 945-953.

Pujol, X. M., 2002. *Image segmentation integrating colour, texture & boundary information,* s.l.: Department of Electronics, Informatics and Automation, Universitat de Girona.

Ratinov, L. & Roth, D., 2009. *Design challenges and misconceptions in named entity recognition.* s.l., s.n., pp. 147-155.

Ratner, A. et al., 2018. *Snorkel: Rapid Training Data Creation with Weak Supervision..* s.l., s.n.

Russakovsky, O. et al., 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision,* 115(3), pp. 211-252.

Santos, C. d., Xiang, B. & Zhou, B., 2015. *Classifying relations by ranking with convolutional neural networks.* s.l., In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics.

Santosh, K., Lamiroy, B. & Wendling, L., 2012. Symbol Recognition using spatial relations. *Pattern Recognition,* pp. 331-341.

Sethi, A. et al., 2017. DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers. *arXiv:1711.03543*.

Shen, Y. & Huang, X., 2016. *Attention based Convolutional Neural Network for Semantic Relation Extraction.* Osaka, Japan, Proceedings of the COLING, 26th Conference on Computational Linguistics.

Simonyan, K. & Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition.. *arXiv:1409.1556*.

Tirkaz, C., Yanikoglu, B. & Sezgin, T. M., 2012. Sketched symbol recognition with autocompletion. *Pattern Recognition,* pp. 3926-3937.

Wang, C., Mouchère, H., Lemaitre, A. & Viard-Gaudin, C., 2017. Online flowchart understanding by combining max-margin Markov random field with grammatical analysis.. *International Journal on Document Analysis and Recognition (IJDAR),* pp. 123-136.

Wimalasuriya, C. & Dou, D., 2010. Ontology based information extraction: An Introduction and a survey of current approaches. *Journal of Information Science*, Volume 36.

Young, T., Hazarika, D., Poria, S. & Cambria, E., 2018. *Recent trands in deep learning based natural language processing,* s.l.: arXiv.org.

1. www.paperswithcode.com [↑](#footnote-ref-2)
2. www.github.com/zziz/pwc [↑](#footnote-ref-3)
3. www.gitxiv.com [↑](#footnote-ref-4)
4. http://yann.lecun.com/exdb/lenet/ [↑](#footnote-ref-5)
5. https://spacy.io/ [↑](#footnote-ref-6)
6. https://stanfordnlp.github.io/CoreNLP/ [↑](#footnote-ref-7)
7. https://www.nltk.org/ [↑](#footnote-ref-8)
8. https://radimrehurek.com/gensim/ [↑](#footnote-ref-9)
9. https://wiki.dbpedia.org/ [↑](#footnote-ref-10)
10. https://www.dbpedia-spotlight.org/ [↑](#footnote-ref-11)
11. https://hazyresearch.github.io/snorkel/ [↑](#footnote-ref-12)
12. https://pythonhosted.org/PyPDF2/ [↑](#footnote-ref-13)
13. https://github.com/tesseract-ocr/ [↑](#footnote-ref-14)