**DARPA ASKE DCC – Milestone Month 15 – (Jan. 2020)**

Akrotirianakis Ioannis, Amar Viswanathan Kannan, Fradkin Dmitriy, Roy Aditi, Tugba Kulahcioglu, Canedo Arquimedes, Mohammad Al Faruque

**Contents**

[1. Introduction 2](#_Toc31405023)

[2. Text2graph 2](#_Toc31405024)

[3. Image2graph 5](#_Toc31405025)

[4. Code2graph 7](#_Toc31405026)

[5. Knowledge Graph 9](#_Toc31405027)

[6. Graph Embeddings 10](#_Toc31405028)

# Introduction

The purpose of this report is to provide an overview of the work that has been done during **December 1st 2019** to **January 31st, 2020**. As the project reaches its completion we will also provide a summary of the main developments and results of our project.

We start by recalling our main motivation. Deep Learning (DL) has experienced a remarkable growth in recent years with thousands of researchers working on new models, architectures and applications every day. DL models are complex to express and implement, and oftentimes require the experts to spend a lot of their time reading through the latest publications, trying to understand the claims, the contributions and the results. Furthermore, machine learning researchers try to use their experience to create links between the body of existing knowledge in order to validate new ideas. However, as DL algorithms and implementations across various fields grow at an explosive rate, keeping up with all the latest publications and their accompanied source code becomes a great challenge for both researchers and practitioners who want to contribute their work and advance the field of deep learning.

The main goal of our project - **Deep Code Curator (DCC)** - aims to address this issue by utilizing the information from scientific publications and the accompanied source code that implements the proposed architectures and methodologies. The DCC collects, extracts and analyzes the scientific papers and represents them in a universal representation that can dramatically decrease the time, effort and resources spent curating deep learning literature and algorithms. We focus on three main modalities that can provide useful information: *text, images,* and *source code*. We have developed three main modules to process each of these modalities: *text2graph, image2graph,* and *code2graph*. The goal of these modules is to create the Resource Description Framework (RDF) Knowledge Graphs (KG) that can be later processed and merged into a super-graph combining the knowledge extracted from all three modalities. This super-graph serves as the architectural representation of a scientific paper and can be used to explore and compare the papers across various fields that share similar deep learning architectures.

In the remaining of this paper we will describe the main modules and functionalities of DCC. More detailed discussions can be found in previous milestone reports.

# Text2graph

We have developed an end-to-end approach that extracts entities and relationships from the text obtained from scientific articles (in PDF format). These articles describe deep learning (DL) models, algorithms, architectures and applications, and they are accompanied by source code which is publicly available. In addition, we have generated a KG that integrates the extracted entities and relations from each article. To achieve our goal we have followed three main steps: (1) annotated the DL papers, (2) develop machine learning models that can predict entities and relations in DL articles that are not annotated, and (3) use the entities and the relations to construct the KG. The general pipeline of the text2graph module is shown in Figure 1.

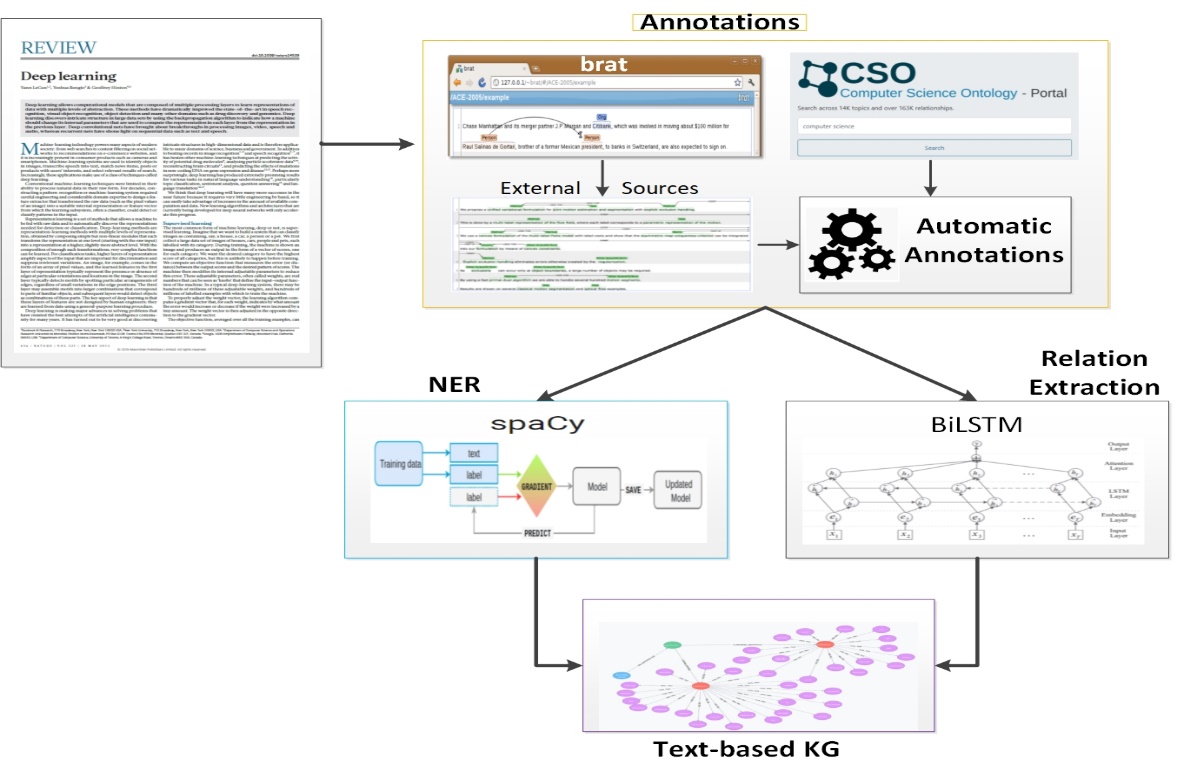


Figure 1: Pipeline for text2graph

The initial annotation of entities and relations in the extracted text was performed manually using the web-based tool Brat[[1]](#footnote-2). Since this process can be very time consuming and difficult to scale up, we have implemented a procedure that is able to automatically produce annotations by utilizing the Computer Science Ontology[[2]](#footnote-3) (CSO). Next, the annotated text is used as input to train statistical models for Named Entity Recognition (NER) and Relation Extraction (RE). These models are used to predict entities and relations in new text that has not been annotated before. The predicted entities and relation are used to construct the knowledge graph of all DL papers we have collected.

We have defined the following entities and relations:

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

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

For NER we have used the spaCy[[3]](#footnote-4) library, mainly due to the following three main advantages: (a) it is considered one of the fastest libraries for large-scale information extraction, (b) it provides very efficient machine learning algorithms for various NLP tasks, and (c) it incorporates easily with major deep learning libraries such as TensorFlow. SpaCy also enables the addition of arbitrary new classes to the NER model. This feature allowed us to easily integrate our six new entities defined above.

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

Finally, we use the extracted entities and relations to build a KG based on the collection of DL documents. The KG is generated by triples of the form <*subject, predicate, object*>, where the *subject* and *object* are extracted by our NER model and the *predicate* is specified by the RE component. Thus, we can view the triple statement as a <*entity1, relationship, entity2*>. For example, in the following sentence[[4]](#footnote-5), "*finally, we can use recurjac to evaluate the robustness of neural networks, by giving a certified lower bound within which no adversarial examples can be found*”, the generated knowledge graph triple would look like:

<Recurjac, used-for, robustness>

Finally, in Figure 2 we show the knowledge graph constructed by extracting entities and relationships from the text of the paper with title: “*PointNetVLAD: Deep Point Cloud Based Retrieval for Large-Scale Place Recognition*”[[5]](#footnote-6). In that graph we use different colors[[6]](#footnote-7) to describe different entities. As can be seen we have identified many entities and relations. For example, “deep networks” and “autoencoder” have been correctly identified as a Method (shown in green circles). Also, “search”, “retrieval”, “registration”, “maximizing” have been correctly identified as Tasks (see the yellow circles).

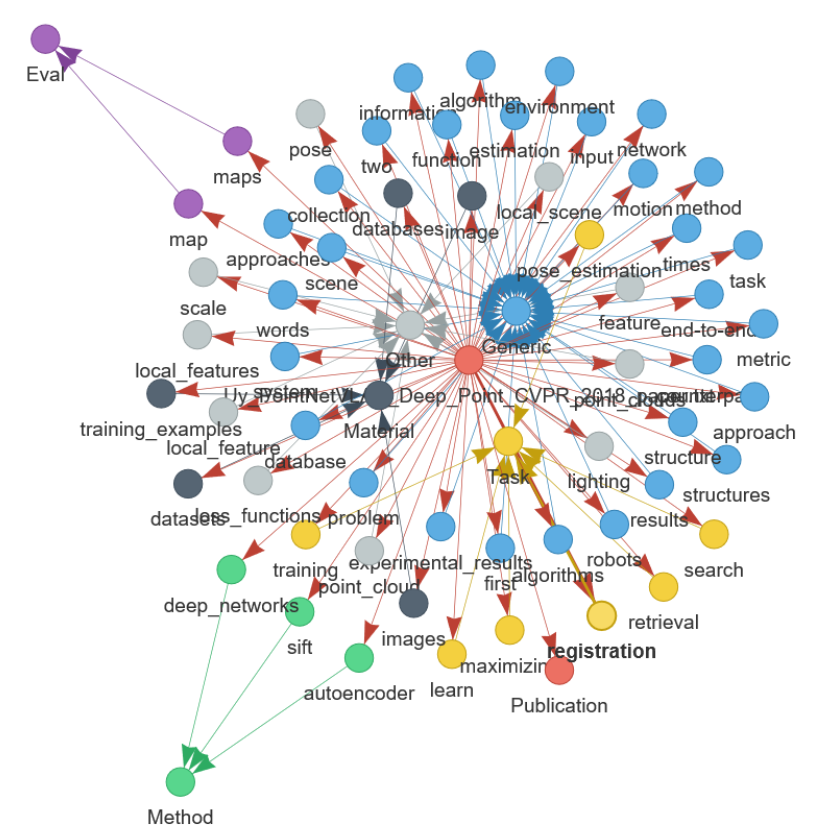


Figure 2: Knowledge Graph constructed from the text extracted from a single paper. Similar graphs are constructed for all papers in our dataset.

# Image2graph

Given the PDF file of a deep learning research paper, the image2graph module consists of four major components: (1) extraction of all the figures from a research paper, (2) identification of those figures showing DL model diagram, (3) analysis of the diagrams depicting DL models, and (4) construction of a knowledge graph representing the information extracted from the DL diagram.

Given the PDF of a deep learning research paper, the image2graph module consists of four major components: (1) extraction of all the figures from a research paper, (2) identification of those figures showing DL model diagram, (3) analysis of the diagrams depicting DL models, and (4) construction of a graph representing the information extracted from the DL diagram.

The general pipeline of the image2graph module is shown in Figure 3.

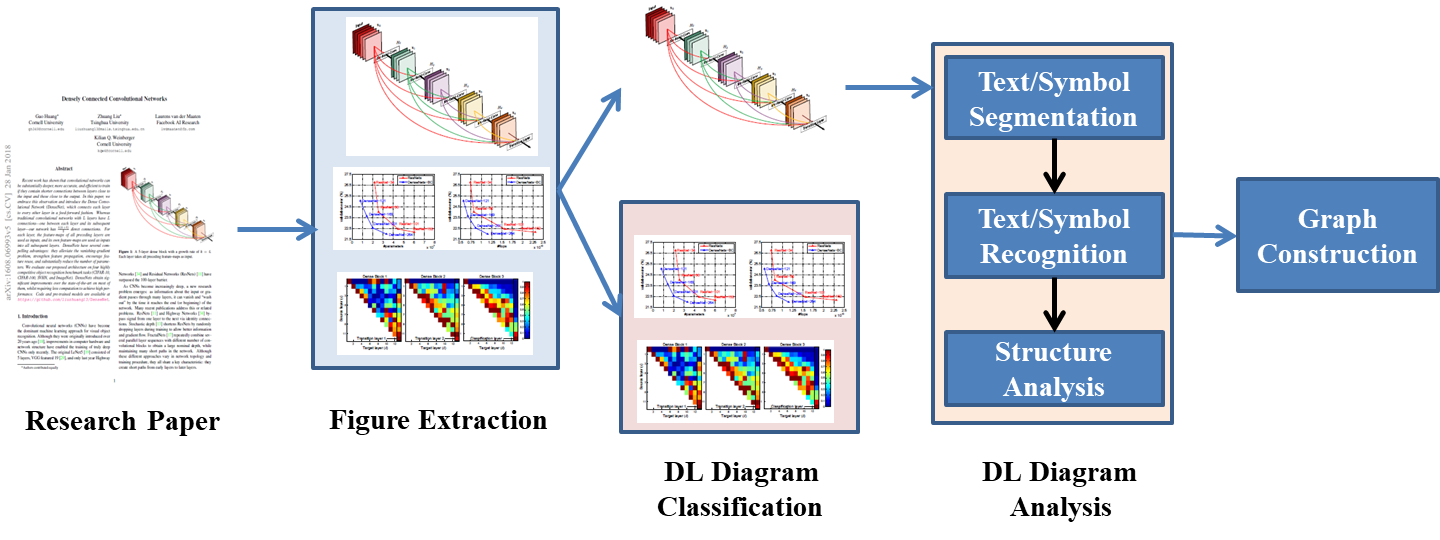
****

Figure 3: Pipeline for image2graph

The image2graph module takes as input the extracted image showing a DL architecture and returns as output a set of relations denoted as

<subject – predicate - object>.

For example, if a DL diagram contains a box that represents an activation function and this activation function, and that activation function is a “leaky ReLu”, then the image2graph will return the following two triples (facts):

1. <component1 – isType - activation>
2. <component1 – hasDescription – [‘leaky’, ‘relu’]>.

Currently we have identified 9 relations to represent a diagram as follows:

1. “*isA*”: <FigureID - isA - Figure>
2. “*foundIn*”: <FigureID - foundIn – “paper title”>
3. “*hasCaption*”: <FigureID - hasCaption – “figure caption”>
4. “*partOf*”: <ComponentID/TextTD - partOf - FigureID>
5. “*hasPos*”: <ComponentID/TextID - hasPos - (x, y, h, w)>
6. “*isType*”: <ComponentID/TextID - isType – “layer name”>
7. “*hasDescription*”: <ComponentID/TextID - hasDescription - [list of words]>
8. “*hasFlow*”: <FigureID - hasFlow – “flow direction”>
9. “*followedBy*”: <ComponentID/TextI - followedBy - ComponentID/TextID>

Each diagram figure is represented by a unique FigureID which is a combination of paper file name and figure number within that paper. The “foundIn” relation represents the title of the paper from where the current diagram is extracted. The relation “hasCaption” represents the caption associated with the current diagram figure. Next, from each diagram multiple *text* components or *node* components are extracted each of which is represented by a unique ID. The relation “partOf” establishes the presence of such unique components. Each component is described with its position using the relationship “hasPos”. The type of the component in terms of different layer name is captured with “isType” predicate[[7]](#footnote-8). All other description found in the text box or node (like kernel size, etc) is put under “hadDescription” predicate.

Finally, to establish links among multiple components found in a diagram, flow direction is an important information. The relation “hasFlow” captures overall flow direction information in terms of either ”left-to-right” or “right-to-left” or “top-to-bottom” or “bottom-to-top”. Individual connectivity between two components is captured using “followedBy” predicate.

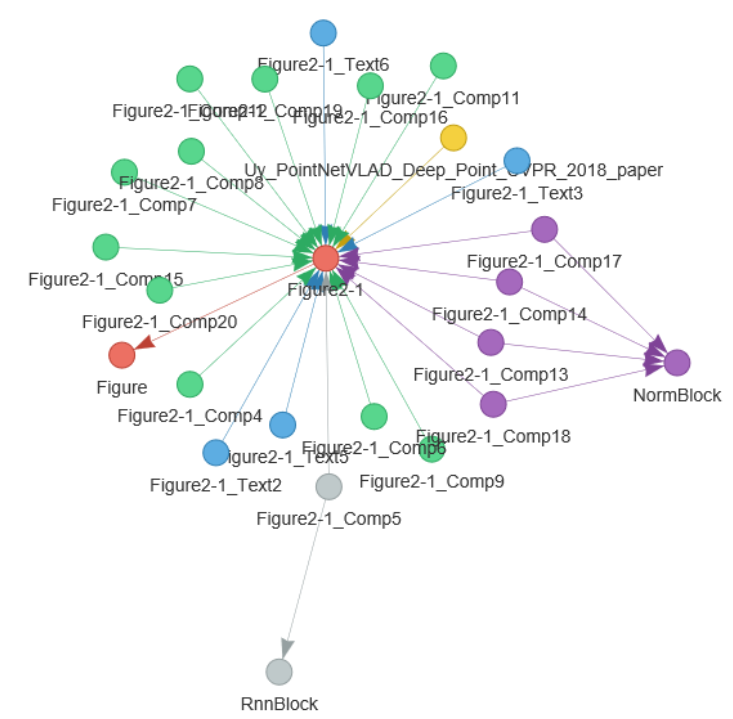


Figure 4: Knowledge graph constructed from a single image showing a DL architecture

# Code2graph

The main aim of the code2graph component is to extract the architectural information from the code repositories that accompany the DL papers we have extracted and considering. To be more specific, knowledge graphs in the form of Resource Description Framework (RDF) graphs are extracted from the source code which needs to be written in Python and uses TensorFlow and/or Keras as the underlying deep learning framework. The main aim in extracting the knowledge graphs from code is two-fold. First, it will serve to complete the information stored in the knowledge graphs of the other two modalities (text2graph and image2graph). Second, the aggregated knowledge graph (i.e., the one that consolidates all modalities – text, images, and code) serves as the super graph from which we perform novelty and similarity detection among other scientific papers.

We have developed two main approached:

1. The Computational Graph-based Approach, which is shown in the green-colored boxes in Figure 5, we created a pipeline to extract and simplify the RDF graphs by executing source code and tracing the path of execution as the program runs.
2. The Lightweight Approach, which is shown in blue-colored boxes, extracts the graphs by analyzing the abstract syntactic structure of the code itself.

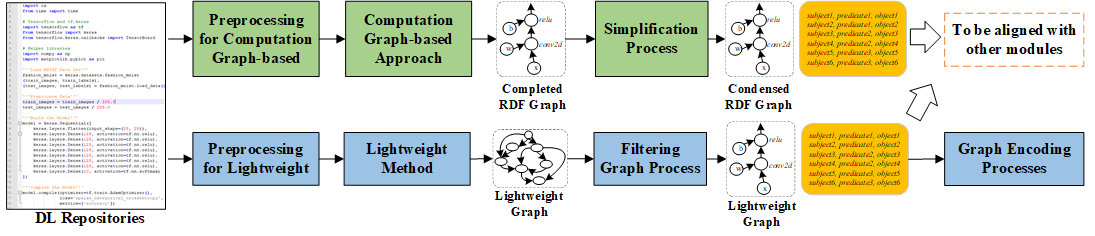


Figure 5: Pipeline for code2graph

We have focused on the extraction of RDF graphs where compilation or running the entire code repository is not necessary (i.e., the Lightweight approach). This approach is more efficient and can scale up to large code bases (i.e., many github repositories representing the source code of DL papers). We exploit the fact that programmers invoke TensorFlow and/or Keras APIs in order to make their scripts simple and easy to debug. Therefore, tracking the occurrence and the order by which TensforFlow and/or Keras functions are invoked can help us to re-construct the computation graph of the DL model that the programmers have in mind. Our approach can be essentially thought of as a *deep code locator* that syntactically traverses the source codes looking for specific keywords used by TensorFlow and/or Keras.

Initially we extract the Abstract Syntax Trees[[8]](#footnote-9) (AST) which consist of important information such as implemented modules and methods in the proposed DL architecture. Utilizing the ontologies and vocabulary, we can extract the deep learning architecture information implemented in the architecture (e.g., layer types, activation functions, optimizers, etc.) In order to get the program structure and architecture information, a static call graph is generated[[9]](#footnote-10) for the project containing the DL source code. After that, we determine the starting points by finding the *zero in-degree vertexes* on the call graph, which means no other modules or functions will invoke this function in the project. Next, we start to inspect the body of the functions and the remaining files in the project. Child nodes in the generated tree structure can be used to identify whether the function calls are *TensorFlow* or *user-defined* function calls.

By inspecting all the source files in a project in this line-by-line manner, we then can build the call trees for the projects. With the call trees extracted, we then generate the RDF graphs to represent a DL model. The algorithm will determine each of the function calls in the call tree to be either *UserDefined* or *TensorFlowDefined* by performing a function call mapping procedure. According to the call tree structure, the corresponding relations (such as *calls* and *followedBy*) between functions will then be added. For all the RDF graphs that have been successfully extracted using the lightweight method, we have updated their format so that they can be processed and parsed by the Ontology viewer Protege[[10]](#footnote-11).

# Knowledge Graph

We have focused on two main areas: (a) the design of our ontology which plays an important role in the development of the KGs for the three different modalities (text2graph, image2graph, and code2graph) and, (b) on the KG alignment.

**Ontology design:** We represent the extracted components from text, images and code segments as instances of a knowledge graph. This allows us to query and visualize the three different modalities of the scientific paper. It also allows us to explore how different modalities interact with one another. The schema of our knowledge graph (DeepSciKG.nt) seeks to represent all the modalities in one single representation. This ontology development focused on utilizing domain expert created RDF and RDFS definitions spanning all three modalities. Our current ontology thus consists of 277 object classes, 24 Object properties, 18 data properties and 2 annotation properties. The ontology schema can be summarized as follows:

* Top level classes:

CodeEntity, FigureComponent, Function, ImageComponent,

Modality, Publication, PublicationAuthor, Repository,

SourceCodeFile, TextEntity, tf

* **CodeEntity** consists of two kinds of classes: TensorFlowDefined and UserDefined.
* **FigureComponent** categorizes all the image2graph extractions as their own classes and they are: *ActivationBlock, ConcatBlock, ConvBlock, DeconvBlock, DenseBlock, DropoutBlock, EmbedBlock, FlattenBlock, InputBlock, LossBlock, LSTMBlock, LSTMSeqBlock, NormBlock, OutputBlock, PoolingBlock, RNNBlock, RNNSeqBlock, UnpoolingBlock.*
* **Function** and **ImageComponent** are used to categorize instances and map them to the right classes.
* **Modality** consists of Code, Figure and Text, where Text is subdivided into AbstractText, BodyText, CaptionText and TitleText
* **Publication** and **PublicationAuthor** focuses on representing just the publication metadata and author metadata
* **Repository** and **SourceCodeFiles** are used for more fine-grained representations of source code data
* **TextEntities** represent the set of classes that are extracted from the text2graph task. We currently extract – Method, Task, GenericTerm, Material, Evaluation Metric, and OtherScientificTerm
* **tf** represents the hierarchy of modules that are present in the tensorflow documentation

Furthermore, the 24 Object properties and the 18 Data properties are used to *link* the instances together in conjunction with the linked data principles. The creation of these properties allowed us to incorporate basic initial **graph alignment per paper**. We extract instances of *text entities, image blocks, and code segments,* and then map them to the right classes in the ontology. This allows us to query all the three modalities *simultaneously*. This extraction of instances according to the ontology classes allows us to create a **rich multimodal knowledge graph** which is present in our repository as [consolidated.ttl](https://osf.io/a76t8/). i.e., the knowledge graph that consolidates all three modalities we consider.

**Graph Alignment:** For graph alignment, we focused on the following two techniques:

* ***Implicit graph alignment:*** We focus on the inferencing capabilities of the knowledge graph and then utilize domain specific properties and rules to link all the modalities of the paper together. We have utilized the unique URIs of all the papers and then linked instances of text, images and code to them accordingly. This allowed us to *query for per paper multi modal graphs*. The queries can be found in our github repository at <https://github.com/deepcurator/DCC/blob/master/queries.py>.
* ***Explicit graph alignment:*** We have decided to use an external well curated knowledge graph called the Computer Science Ontology (CSO). The CSO is a large scale research ontology that categorizes 16 million publications, mainly in the field of computer science as a taxonomy. Out of the 14K topics and 163K relationships, we mapped all our instances of concepts from the text extraction (i.e., the entities of the text extraction to their respective CSO components). The entities that were mapped were instances of Material, Method, Metric and Task. This amounted to a total of 978 statements.

# Graph Embeddings

In this section we briefly discuss the key ideas of graph embeddings that we have used in our project. We assume that a KG has been constructed[[11]](#footnote-12) and represented by the graph G=(V,E), where V is the set of nodes (i.e., entities) and E is the set of edges (i.e., relationships between entities). We will denote the cardinality of the set of nodes by . We also assume that a predefined lower dimension *d* (with << ) is given. The problem of *graph embedding* is defined as the process of converting G from its original -dimensional space to a lower -dimensional space in such a way that certain *proximity measures are preserved*. In the lower dimensional space (also referred as the embedded space) the original KG is represented as a -dimensional vector, or a set of -dimensional vectors with each vector representing the embedding of a part of the KG (e.g., nodes, edges, substructures, etc.)

In recent years a large number of graph embedding techniques has emerged[[12]](#footnote-13). These techniques share two main common mapping functions (i.) an *encoder*, whose aim is to map each node of the original graph to a lower-dimensional vector space (embedding), and (ii.) a *decoder*, which tries to recover or reconstruct structural information about the original graph from the learned embeddings. Once the learning process is complete, we can use the embedding space (which is the output of the encoder) as feature inputs to perform various machine learning tasks. For example, we could input the embedding vectors to a logistic regression classifier and predict the group that a node belongs.

The recent success of graph embeddings is attributed to the use of *deep autoencoders*[[13]](#footnote-14) which unify the encoder and decoder parts in such a way so that information about the local neighbors of each node is compressed in a lower dimensional space. More specifically, each node, , of the KG is associated with a vector, , that can store the proximity information of the current node with its neighbors. A commonly used proximity measure is the adjacency matrix[[14]](#footnote-15) of the KG. The main aim of the *graph autoencoder[[15]](#footnote-16)* (GAE) is to embed the KG nodes using the proximity vectors in such a way that they can be reconstructed from their embeddings, that is, DECODER(ENCODER()) ~= , where ENCODER() = represents the lower dimensional embedding of the KG that we want to learn. The encoder and decoder that constitute the GAE are implemented by various types of deep neural network layers. Each layer of the encoder reduces the dimensionality of its input, while each layer of the decoder increases it.

**Graph embeddings for code2graph:** We analyzed the source code associated with a DL publications and extracted the information into an intermediate form that can capture the high-level semantics from DL publications. To do this, we represent the *code elements* (e.g., functions defined by the developer of the source code) using low-dimensional latent vectors (embeddings). The latent vector captures the semantic properties of an object by representing it with its features distributed across multiple vector components. To learn the representations, we have used our own library pykg2vec[[16]](#footnote-17) which includes a collection of well known KG embedding models as well as tools for evaluating them. By leveraging pykg2vec, we are able to apply multiple translational-based and similarity-based KG embedding algorithms and compare the results for our datasets. In code2graph, a given graph is essentially a *function call graph* and is treated as a KG containing nodes that represent either *internal function calls* or *TensorFlow function calls* and edges that represent the *calling* or *following* relationships.

The details of the training process are described in the report of Milestone M13. Here will only discuss the results obtained when using the T-SNE[[17]](#footnote-18) embedding visualizer as shown in Figure 6. The visualization demonstrates the distribution of all the entities (i.e., TensorFlow\_functions, Userdefined\_functions, Modules). From Figure 6, we can also see that the clustering of entities shows good results. For example, Userdefined\_functions are staying together on the left side of the embedding space. This demonstrates that, with KG embedding methods alone, it is possible to capture the semantic meaning of source code and the entities on the Lightweight Graphs.

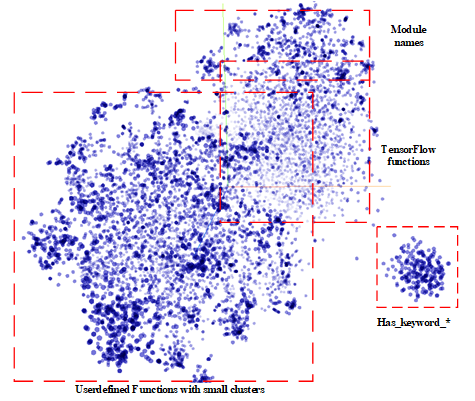


Figure 6: Visualization of all the function node embeddings with TransE.

**Graph embeddings for all modalities:** The graph embedding approach is applicable to all three modalities we work with. We leverage the Graph AutoEncoder (GAE) approach to accomplish subgraph embedding. The graph embedding proceeds as described above for all the edges and nodes, with one difference: we introduce a *super-node* for each subgraph (i.e., each paper/image/repository) that is connected to all the nodes in that subgraph and serve as subgraph representative. Thus, when embedding is done, *the super-node embedding can be seen as a summary of the subgraph*. For the visualization purposes we take the embedding of the super-nodes and apply dimensionality reduction such as TSNE to generate the images shown in Figure 7 below. Note that ECCV, CVPR are conferences that specialize in image processing and computer vision. This can be observed in Figure 7 (left plot) where we see in the bottom-left part a clustering of green (CVPR) and orange (ECCV) points. On the other hand, NIPS covers more broad deep learning topics (not necessarily related to computer vision). Similarly, ICML is still broad, since it generally includes papers that are not directly related to Deep Learning. The characteristics of the data used in these embeddings are summarized below:

* *Text:* we have graphs from 653 papers, and they comprise of 13,770 nodes with 733 features per node and 47,112 edges/relations between them.
* *Images:* we constructed graphs for DL architecture images from 143 of the above papers. They comprise of 4,252 nodes with 393 features per node and 16,098 edges/relations between them.
* *Code:* we were unable to run this specific approach on the complete data from 345 repositories, since that included several 100K nodes and >2M relations. For this reason, we focused on a smaller set of 63 repositories with lightweight graphs. The data comprised 8,016 nodes and 56,107 edges. We used topics as color labels (see Figure 8).

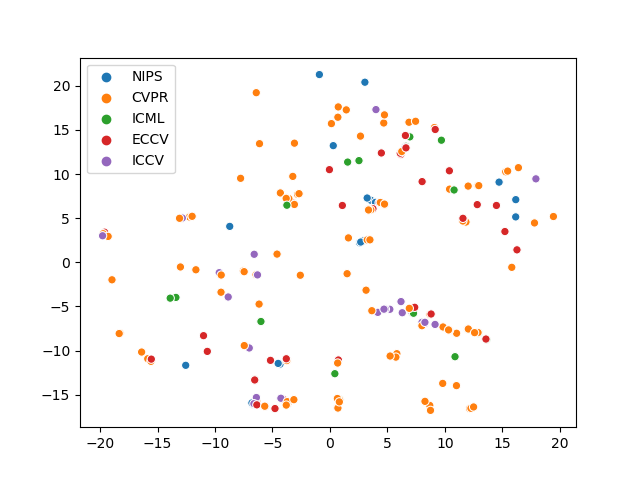
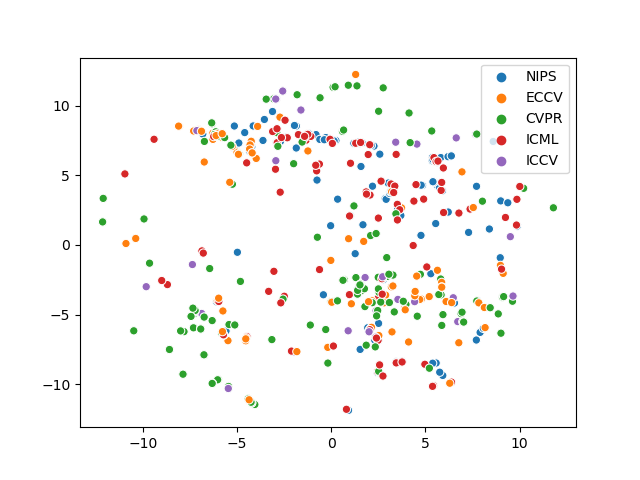


Figure 7: Embeddings based on abstract text of the 653 papers (left plot) and on 213 images from 143 of the same papers (right plot). The colors are based on conferences where the papers have been pulished.

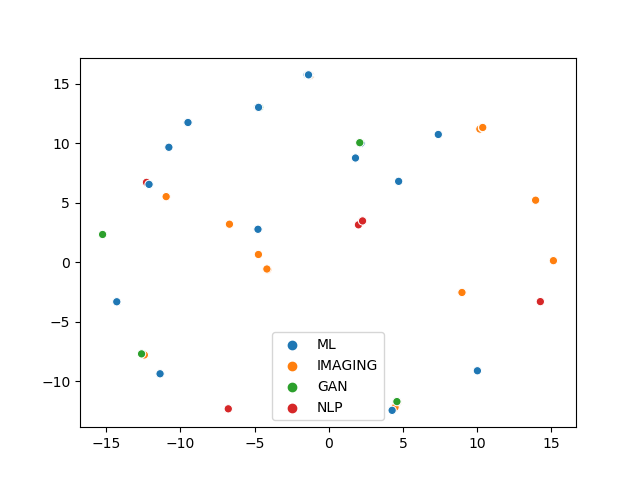


Figure 8: Embeddings of graphs from 63 code repositories

1. http://rat.nlplab.org [↑](#footnote-ref-2)
2. <http://skm.kmi.open.ac.uk/cso/> [↑](#footnote-ref-3)
3. <https://spacy.io> [↑](#footnote-ref-4)
4. Taken from the paper: H. Zhang, P. Zhang, CJ. Hsieh “RecurJac: An efficient recursive algorithm for bounding the Jacobian matrix of neural networks and its applications” <https://arxiv.org/abs/1810.11783> [↑](#footnote-ref-5)
5. <https://arxiv.org/abs/1804.03492> [↑](#footnote-ref-6)
6. Green for Methods, yellow for Tasks, Blue for Generic, Black for Material, Purple for Evaluation metric, and Grey for Other. [↑](#footnote-ref-7)
7. The layer names used for “isType” predicate are: "conv, "deconv”, "dense”, "flatten”, "dropout", "pooling", "unpooling", "concat", "RNN", "RNNseq", "LSTM", "LSTMseq", "normBlock", "embed", "activation", "loss". [↑](#footnote-ref-8)
8. Abstract Syntax Tree is a data structure which captures the syntactic structure of the source code for the specific programming language. An AST data structure is normally used by compilers to represent the structure of the program [↑](#footnote-ref-9)
9. Using the pyan package - <https://github.com/davidfraser/pyan> [↑](#footnote-ref-10)
10. <https://protege.stanford.edu/> [↑](#footnote-ref-11)
11. For example, the KGs constructed by the text2graph, image2graph and code2graph modules. [↑](#footnote-ref-12)
12. H. Cai, V. W. Zheng, and K.C.C. Chang, “A comprehensive survey of graph embeddings: problems, techniques and applications”, <https://arxiv.org/abs/1709.07604> [↑](#footnote-ref-13)
13. I. Goodfellow and Y. Bengio and A. Courville, “Deep Learning”, MIT Press, 2016 [↑](#footnote-ref-14)
14. The adjacency matrix of a graph is defined as the matrix whose element is equal to 1 if there is an edge connecting vertex with vertex . In the opposite case, the element of the adjacency matrix is equal to zero. [↑](#footnote-ref-15)
15. T.N. Kipf and M. Welling, “Variational graph auto-encoders”, <https://arxiv.org/abs/1611.07308> [↑](#footnote-ref-16)
16. Yu, Shih Yuan, et al. "Pykg2vec: A Python Library for Knowledge Graph Embedding."; <https://arxiv.org/abs/1906.04239> [↑](#footnote-ref-17)
17. L. van der Maaten, “Accelerated t-SNE using tree-based algorithms”, Journal of Machine Learning Research, 15, pp. 1-21 (2014); <http://lvdmaaten.github.io/publications/papers/JMLR_2014.pdf> [↑](#footnote-ref-18)