**DARPA ASKE DCC – Milestone Month 18 – (Apr. 2020)**

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

**Contents**

[1. Introduction 2](#_Toc39179919)

[2. Text2graph 2](#_Toc39179920)

[3. Image2graph 5](#_Toc39179921)

[4. Code2graph 7](#_Toc39179922)

[5. Knowledge Graph 11](#_Toc39179923)

[6. Paper2Graph 13](#_Toc39179924)

[7. Graph Embeddings and Insights 14](#_Toc39179925)

# Introduction

This report provides an overview of the work that has been done during Phase II of the DARPA ASKE program. The period covered is from **December 1st 2019** to **April 30th, 2020**.

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. We will be using an example DL paper[[1]](#footnote-2) and its accompanied GitHub source code[[2]](#footnote-3). The example DL paper will help us demonstrate the results obtained by text2graph, code2graph and code2graph, as well as the super-KG that merges the three modality specific KGs.

The software implementation of DCC is publicly available and can be downloaded from our GitHub repository <https://github.com/deepcurator/DCC/tree/master>. We have included information about the structure of the code and instructions on how to install the various components and 3rd party software packages needed to run it a personal computer. In addition we have included three demos in the form of Python notebooks <https://github.com/deepcurator/DCC/tree/master/demo>. The first demo showcases the KGs generated for the three different modalities and the merged super-KG. The second demo uses the information in the merged KG to answer specific queries. Finally, the third demo shows several embeddings that can be used to generate useful insights.

# 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 GitHub. 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) developed 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.

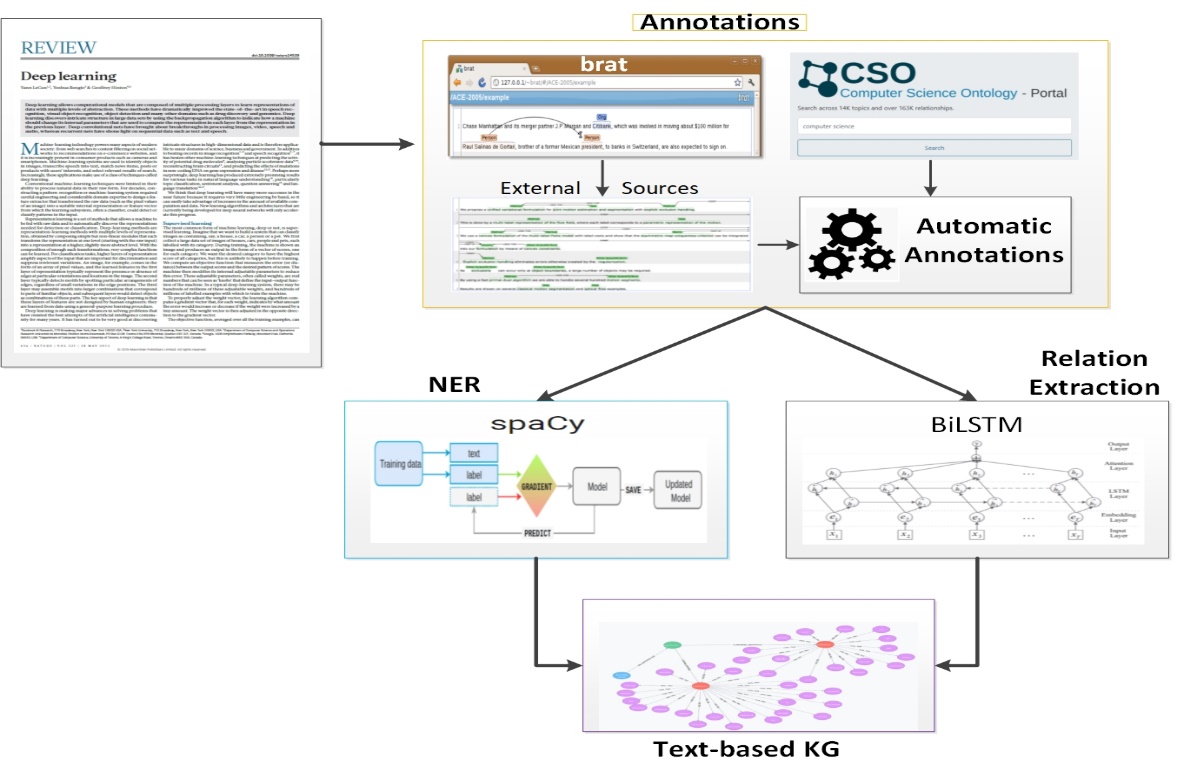


Figure 1: Pipeline for text2graph

The initial annotation of entities and relations in the extracted text was performed manually using the web-based tool Brat[[3]](#footnote-4). 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[[4]](#footnote-5) (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 relations are used to construct the knowledge graph of all DL papers we have collected.

We have defined the following entities and relations:

***Entities*:**

1. **Task** – can be an application, a system that neds to be constructed, or a specific problem to be solved. For example: information extraction, forecasting, image analysis, classification, forecasting.
2. **Method** – can be a model, an algorithm, a system that is used in a scientific paper, or a framework. For example: Neural Network, Attention, CRF, CNN, RNN, optimization, stochastic gradient descent.
3. **Evaluation metric** – can be a set of measurements, quantities, measures. For example: F1, Precision, Recall, ROC curve,
4. **Material** – can be data, resources, images, pdf documents, or a knowledge base. For example: data, data-sets, corpus, MNIST, ImageNet.
5. **Other scientific terms** – can be any scientific term that is not directly related to the topic of Deep Learning (it covers mostly contextual information related to the application a DL methods) e.g., dbpedia, Wikipedia, CoNLL, combustion, earth, physics, bioinformatics, etc.
6. **Generic** – can be any general term that may refer to an entity but it does not add any useful information. For example: model, approach, algorithm, them, they, etc.

***Relations*:**

1. **Used for – e.g.,** B is used for A, B models A,
2. **Feature of -** B belongs to A B is a feature of A,
3. **Part of –** e.g., Our system includes models A and B,
4. **Compare** - Comparing two works,
5. **Conjunction** - Symmetric relation,
6. **isA** – e.g., DNN is a type of artificial neural network, and
7. **sameAs** : ex. NMT, otherwise known as neural machine translation.

For NER we have used the spaCy[[5]](#footnote-6) library, mainly due to the following three main advantages: (a) it is considered one of the fastest libraries for large-scale information extraction, (b) it provides very efficient machine learning algorithms for various NLP tasks, and (c) it incorporates easily with major deep learning libraries such as TensorFlow. SpaCy also enables the addition of arbitrary new classes to the NER model. This feature allowed us to easily integrate our six new entities defined above.

For Relation Extraction 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.

We use the extracted entities and relations to build a Knowledge Graph based on the collection of the DL documents we have gathered. 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[[6]](#footnote-7),

"*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 takes the form:

<Recurjac, used-for, robustness>

Finally, in Figure 2 we show the constructed KG by extracting entities and relationships from the text obtained from our example DL paper[[7]](#footnote-8). In that graph we use different colors[[8]](#footnote-9) 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 (shown in yellow circles).

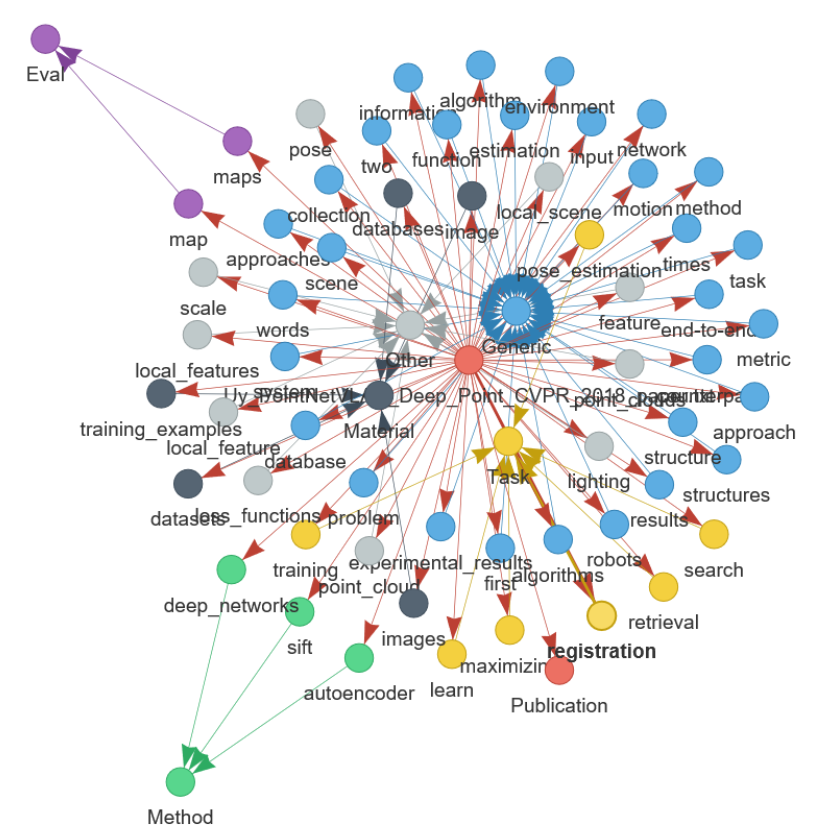


Figure 2: Knowledge Graph constructed from the text extracted from a single paper. Similar graphs are constructed for all papers in our dataset.

# 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 architectures (all the remaining figures are ignored),
3. Analysis of the figures depicting DL architectures, and
4. Construction of a knowledge graph representing the information extracted from the DL diagram.

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

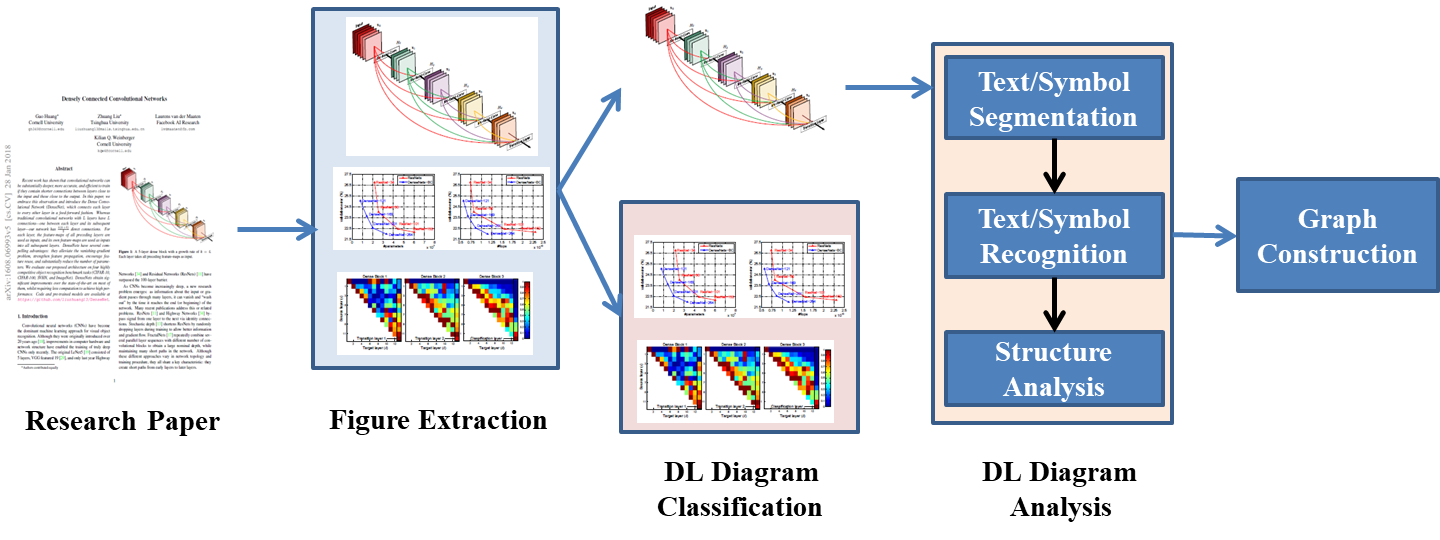
****

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 triples denoted as

<subject – predicate - object>.

For example, if a DL diagram contains a box that represents an 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/TextID - 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[[9]](#footnote-10). 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 shows the KG produced by the image2graph module for our example paper[[10]](#footnote-11). We can see that it has identified the Figure 2 (in the example paper) contains the definition of the DL architecture that the authors of that paper propose. Within that figure image2graph has identified an RnnBlock and a NormBlock (normalization block). Next we can see that the RnnBlock is identified as a recurrent neural network, and the components of the NormBlock are connected together. The green nodes represent text that is captured in the figure and withing the image depicting the DL architecture.

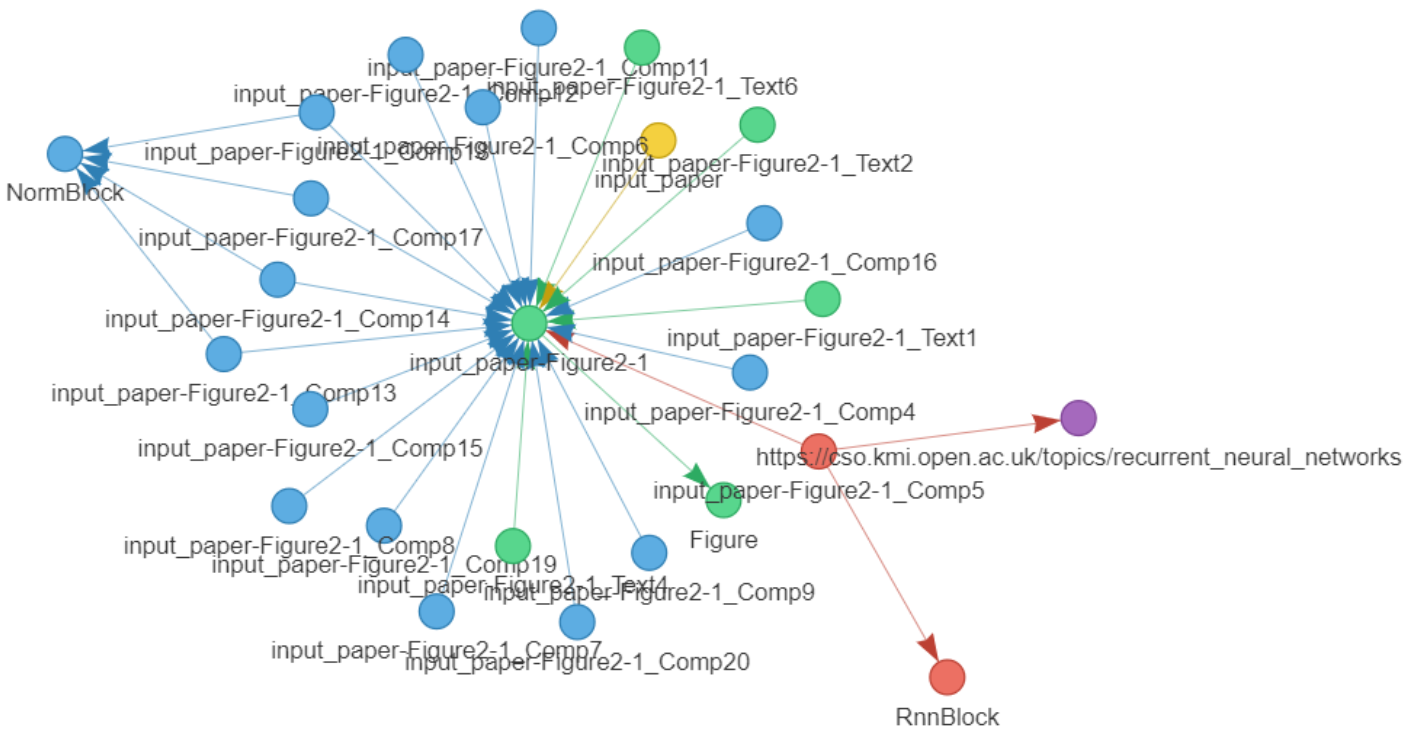


Figure 4: Knowledge graph constructed from a single image showing a DL architecture

# 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[[11]](#footnote-12) and/or Keras[[12]](#footnote-13) 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 DL papers.

We have developed two main approaches:

1. The *Computational Graph-based Approach*, which is shown in the green-colored boxes in Figure 5, which extracts and simplifies 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.

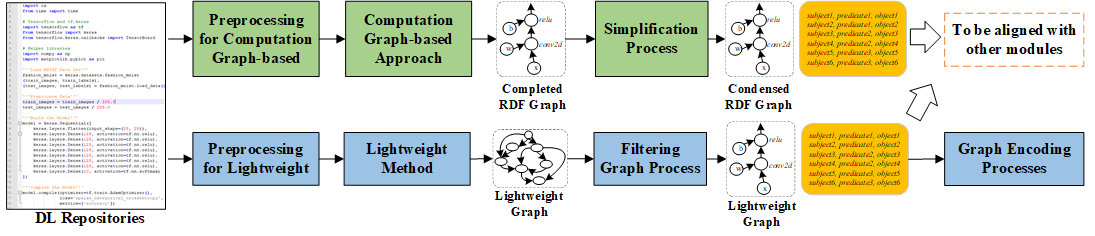


Figure 5: Pipeline for code2graph

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., GitHub repositories containing many lines of code which represent the implementation 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[[13]](#footnote-14) (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 (e.g., layer types, activation functions, optimizers, etc.) In order to get the program structure and architecture information, a static call graph is generated[[14]](#footnote-15) 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 *TensorFlowDefined* or *UserDefined* function calls.

By inspecting all the source files in a source code 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[[15]](#footnote-16).

An example of a KG obtained by calling our code2graph module using as input the source code extracted from the GitHub repository <https://github.com/mikacuy/pointnetvlad> (which corresponds to our example paper[[16]](#footnote-17)) can be seen in Figure 6. The green and yellow nodes in the graph represent the *TensorFlowDefines* and *UserDefined* functions used in the code. A close up of the graph shown in Figure 6, can be seem in Figure 7, where we have focused on the *UserDefined* functions with names train\_3 and best\_pos\_distance\_1. As can be seen we have two types of arrows. The solid arrows represent a “call” relation (e.g., the *UserDefined* function train\_3 “calls” the *TensorFlowDefined* functions AdamOptimizer and the *UserDefined* function placeholder\_inputs\_4). On the other hand, the dashed arrows represent “followedBy” relations (e.g., the *UserDefined* function best\_pos\_distance\_1 is “followedBy” the *TensorFlowDefined* function tile).

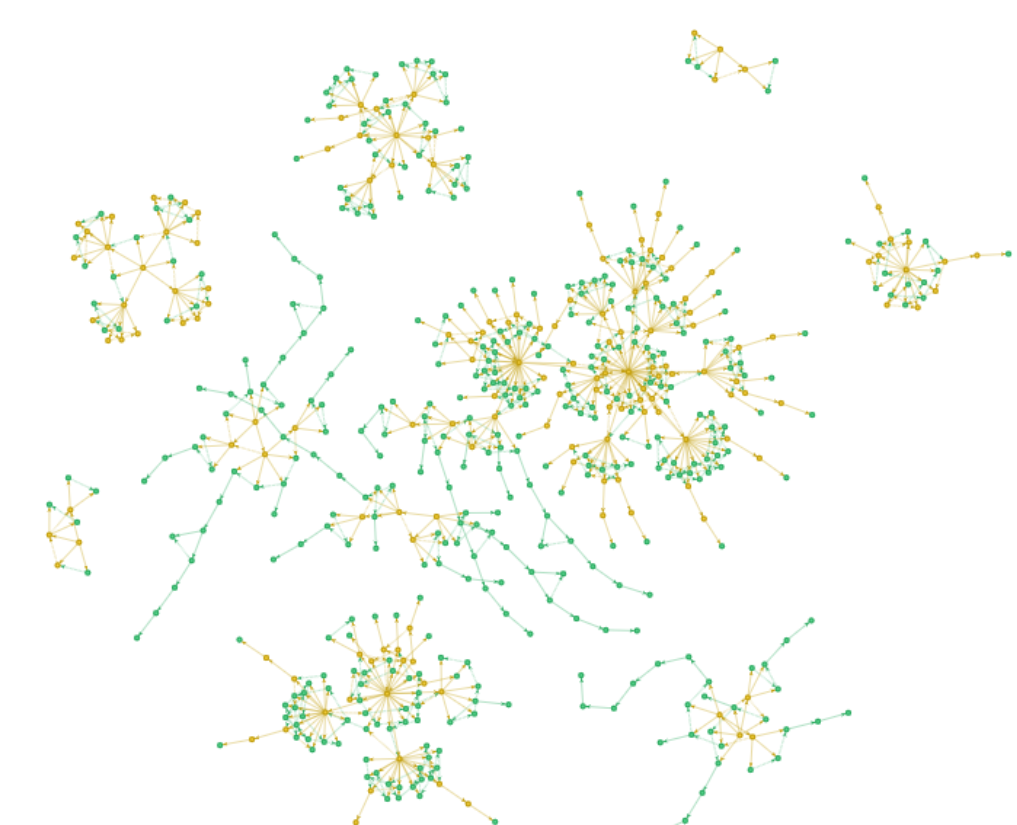


Figure 6: KG obtained by our code2graph module for the source code of the github repository[[17]](#footnote-18)

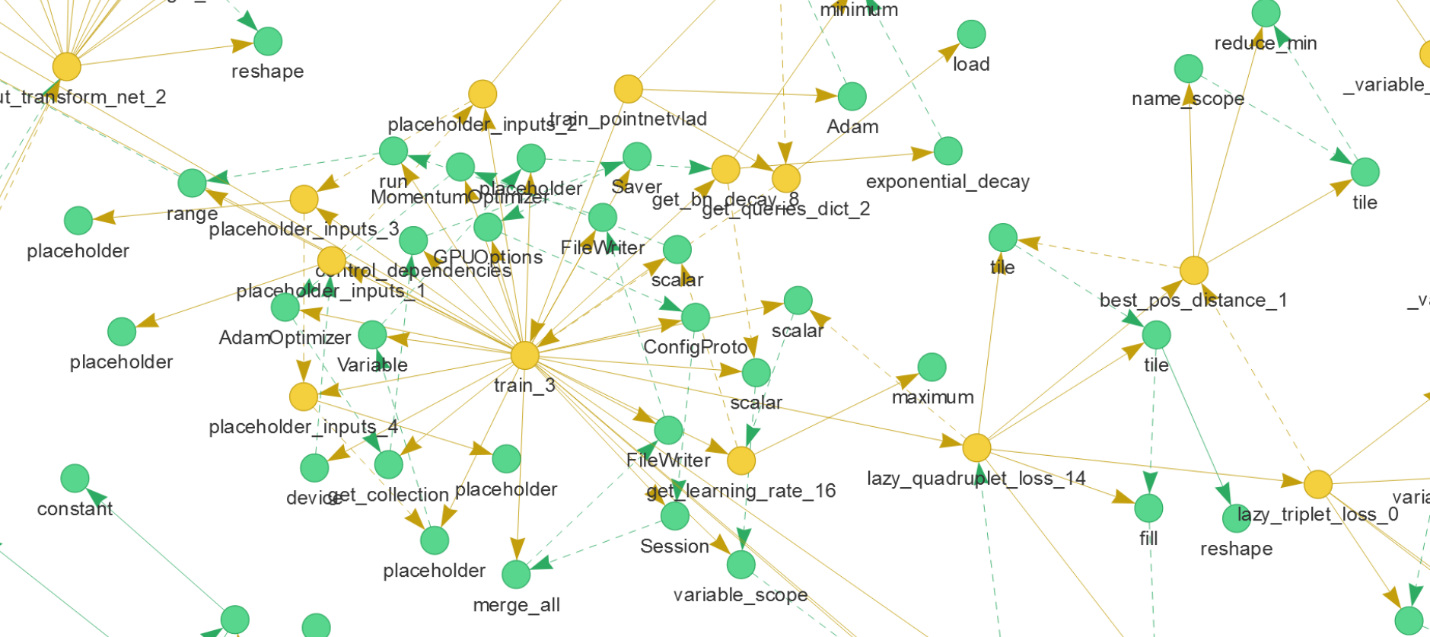


Figure 7: a close up of the code2graph described in the previous figure

# 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 a Deep Learning 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 expertise created RDF and RDFS definitions spanning all three modalities. Our current ontology thus consists of 278 object classes, 25 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. The list of classes we have considered 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 our three main modalities, i.e., Code, Figure and Text. Note that Text is subdivided into: *AbstractText, BodyText, CaptionText* and *TitleText*, which provide information about the place in the paper where the Text was captured.
* **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 module. We currently extract the following entities: *Method, Task, GenericTerm, Material, EvaluationMetric*, and *OtherScientificTerm.*
* **tf** represents the hierarchy of modules that are present in the official TensorFlow documentation[[18]](#footnote-19).

A close-up of the ontology is shown in Figure 8. At the top of the graph we can see the top-level class FigureComponent and next to it the image2graph specific class. Next to NormBlock we can see the four instances (t\_paper\_Figure2-1\_Comp14, …13, …17, …18) which are identified in the image extracted from our example paper[[19]](#footnote-20) and depicts a DL architecture *NormBlock* (note that we saw NormBlock in Figure 4, where we were discussing image2graph). In addition, at the right side of the figure we can see the tf class *TensorFlowDefined* which connects all the TensorFlow functions.

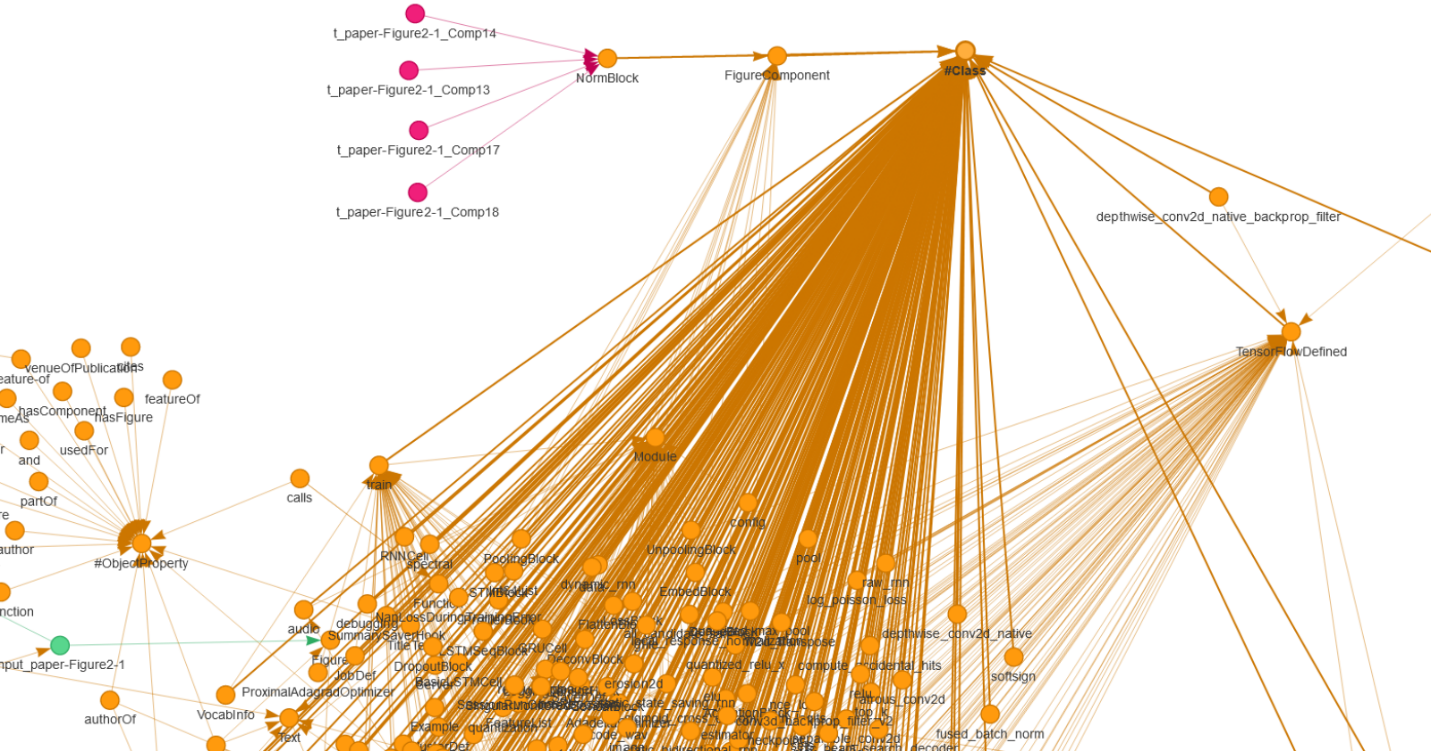


Figure 8: Close up of the Ontology for describing Deep Learning scientific papers and their accompanied source code

**Knowledge Graph population:** 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**. Going back to Figure 8 again, we can see that ObjectProperty class and its constituents (e.g., usedFor, partOf, featureOf, hasComponent, etc.) 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.

The knowledge graph population details for each modality are described next:

*text2graph:*The extracted outputs from the text knowledge extraction components are mapped to the TextEntities mentioned in the previous section. For each of the entities present in the textual content of the paper, we create instances of the entity type and map it to the respective class. For example, a piece of text “Convolutional Neural Network.” would be tagged as “Method” by the Named Entity extractor and this would be mapped as an instance of class Method, having a text label as “Convolutional Neural Network”. In addition, the meta data associated with the paper such as Year, ConferenceType and PaperID are also added to the knowledge graph.

*image2graph:*In the image to graph segment, the components extracted from the image are also mapped to their respective classes the same way as discussed above in the text to graph. The image also contains the paper id which is common to the text.

*code2graph:*The code to graph also maps the variables, parameters, user defined functions and TensorFlow defined functions as instances of a class to the ontology. The code also contains the paper id, which is common to both text and image.

**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 the 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:*** As discussed in section 2 (text2graph), we have decided to use an external well curated knowledge graph called the Computer Science Ontology[[20]](#footnote-21) (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 Method, Task, Material, and Metric. This amounted to a total of 978 statements.

**Discussion on queries:** In this section we discuss a sampling of the queries shown in the demo folder. The last few queries showed how we can use this knowledge graph to aid scientists and scholars. We believe that queries such as “*Finding all the methods used to solve a machine learning task”* from papers across conferences are very useful to see how our system can help find useful machine learning techniques from literature. In addition to finding useful machine learning techniques, this can also be extended to find the respective code entities as well. Now the code entities could be TensorFlow functions and commonly used parameters for these machine learning tasks. This in turn helps us find the *commonalities* across papers working on the same task.

# Paper2Graph

Based on the discussion in the previous sections, we have generated the KG that merges and aligns the three different modality-specific KGs (text2graph, image2graph, and code2graph). We call the merged KG *Paper2Graph*, as it brings together the three major modalities that can be associated with a Deep Learning paper, i.e., text, images and source code.

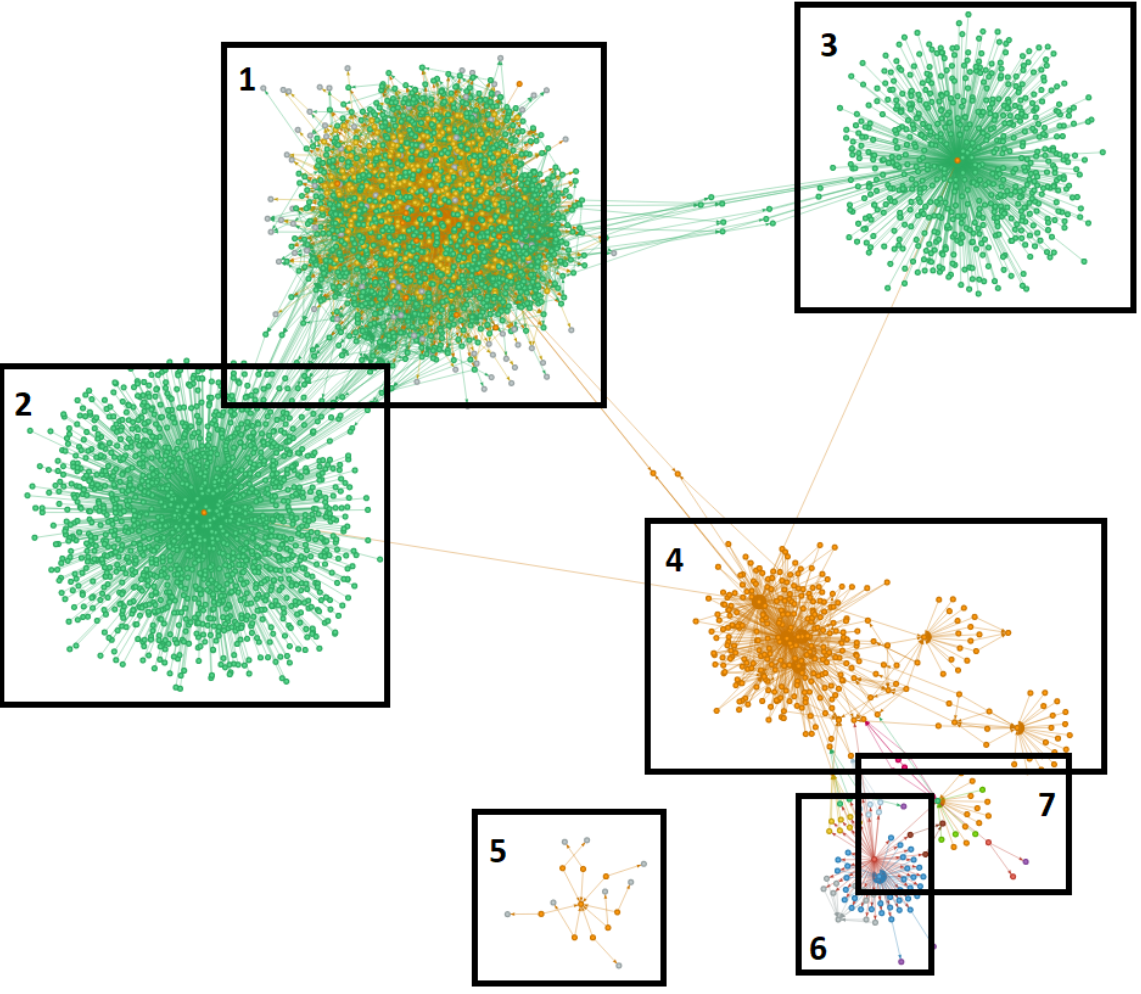


Figure 9: The merged and aligned KG that brings together the three different modalities we consider: text (box 6), image (box 7) and source code (box 1). The ontology developed for our DCC project is shown in boxes 2, 3 and 4, and its main aim is to define the various entities and relations relevant to Deep Learning papers (text and images) and their corresponding source code.

In Figure 9 we have marked the different subgraphs by numbers, the nodes by different colors based on the entities they represent, and the edges by different arrows. More specifically, sub-graphs shown in boxes 1, 6 and 7 represent code2graph, text2graph and image2graph, respectively. Boxes 4, 3, and 2 define our overall Ontology. In particular, Box 4 defines Ontology concepts related to the individual modalities of DCC (text, image and source code). On the other hand, boxes 3 and 2 define the Ontology concepts related to TensorFlow. Box 2 defines the TensroFlow defined functions[[21]](#footnote-22) and box 3 defines the TensorFlow classes[[22]](#footnote-23).

# Graph Embeddings and Insights

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[[23]](#footnote-24) 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[[24]](#footnote-25). 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*[[25]](#footnote-26) 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[[26]](#footnote-27) of the KG. The main aim of the *graph autoencoder[[27]](#footnote-28)* (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[[28]](#footnote-29) 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[[29]](#footnote-30) embedding visualizer as shown in Figure 10. The visualization demonstrates the distribution of all the entities (i.e., TensorFlow\_functions, Userdefined\_functions, Modules). From Figure 10, 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.

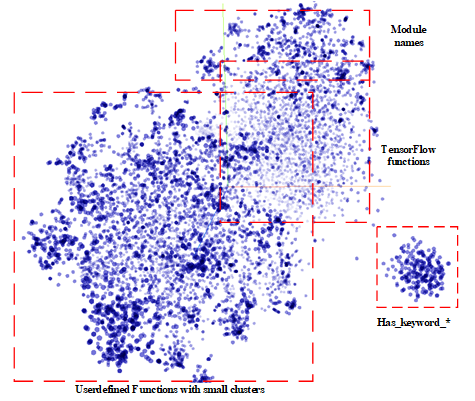


Figure 10: Visualization of all the function node embeddings with TransE.

**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 plots shown in Figures 11-13. The characteristics of the data used in these embeddings are summarized below:

* *Text:* we have graphs from 410 papers, and they comprise of 5,480 nodes with 335 features per node and 17,325 edges/relations between them (Figure 11)
* *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 (Figure 12)
* *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 9,743 nodes and 2305 features and 52,191edges. We used topics as color labels (see Figure 13).

As we can observe in Figure 11 papers from the CVPR (shown in green) and ECCV (shown in blue) conferences tend to cluster together at the top and bottom left sides, respectively. This verifies our intuition since both conferences specialize in image processing and computer vision. On the other hand, since NIPS and ICML cover more broad deep learning topics (not necessarily related to computer vision) can be seen to be more spread and cover a more broad area in the plot.

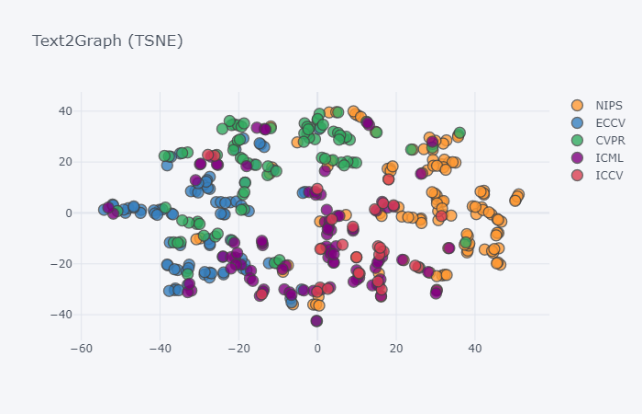


Figure : Embeddings based on text extracted from the abstracts of 410 DL papers.

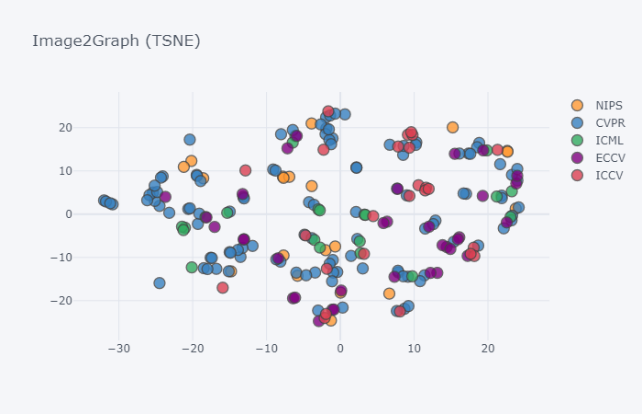


Figure : Embeddings based on 213 images obtained from 143 DL papers (note: these papers are taken from the same DL described in the previous figure)

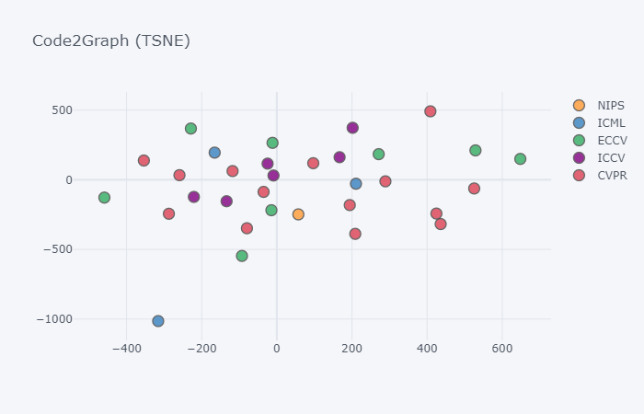


Figure : Embeddings of graphs from code repositories

**Generating insights:** The embeddings we have generated can be used to generate useful insights for the scientific papers we have considered so far. For example, in Figure 14 we are interested in identifying all DL papers in our dataset that contain the entities “*GAN*” or “*Generative adversarial Network*” in their title. These papers are marked with a blue color. Our embeddings successfully capture similarities in terms of the “Method” or the “Topic” that a paper is dealing with. We can see that the GAN papers on the right side are more application oriented and in fact all are from the CVPR conference (a major image processing conference). Two of them[[30]](#footnote-31) seem to be very *similar* since they both focus on enhancing photos. Using this information we can start the process of identifying similar code segments in their respective GitHub repositories, or similarities in their parameter settings (both in the source code and in the text of the paper). The left side in the embeddings contains more theoretical papers as can be seen by the titles and the conferences they have been published (e.g., NIPS and ICML).

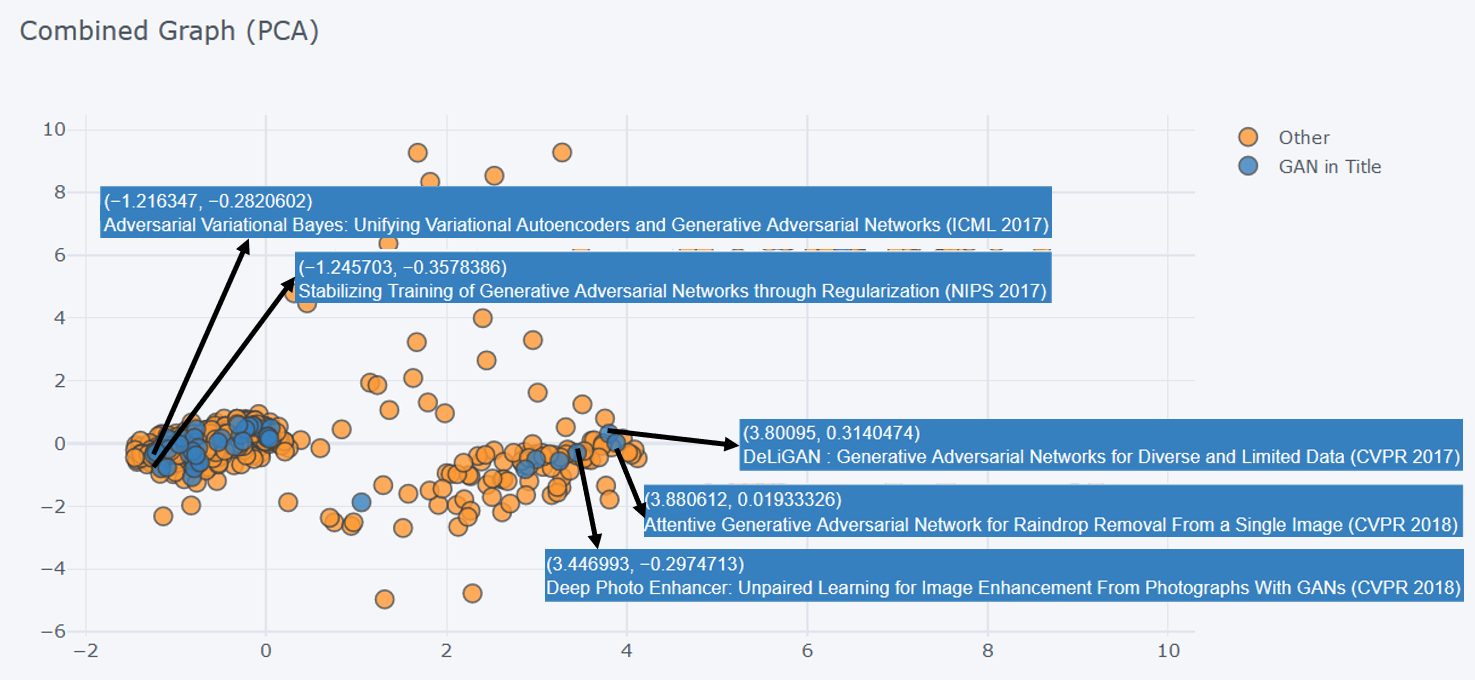


Figure : Identifying similar papers based on a Deep Learning topic or method – in tis example we are interested in identifying similar papers that contain the words “GAN” or “Generative Adversarial Networks” in their title. Papers on the right side deal with applications, whereas papers on the left side discuss more theoretical and algorithmic aspects of GANS as can be seen from their titles and the conferences they have been published.

1. Uy M.A., and Lee G.H., “PointNetVLAD: Deep Point Cloud Based Retrieval for Large-Scale Place Recognition”, CVPR, 2018 - <https://arxiv.org/pdf/1804.03492.pdf> [↑](#footnote-ref-2)
2. <https://github.com/mikacuy/pointnetvlad> [↑](#footnote-ref-3)
3. <http://rat.nlplab.org> [↑](#footnote-ref-4)
4. <http://skm.kmi.open.ac.uk/cso/> [↑](#footnote-ref-5)
5. <https://spacy.io> [↑](#footnote-ref-6)
6. 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-7)
7. <https://arxiv.org/abs/1804.03492> [↑](#footnote-ref-8)
8. Green for Methods, yellow for Tasks, Blue for Generic, Black for Material, Purple for Evaluation metric, and Grey for Other. [↑](#footnote-ref-9)
9. 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-10)
10. <https://arxiv.org/pdf/1804.03492.pdf> [↑](#footnote-ref-11)
11. <https://www.tensorflow.org/> [↑](#footnote-ref-12)
12. <https://keras.io/> [↑](#footnote-ref-13)
13. 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-14)
14. Using the pyan package - <https://github.com/davidfraser/pyan> [↑](#footnote-ref-15)
15. <https://protege.stanford.edu/> [↑](#footnote-ref-16)
16. <https://arxiv.org/pdf/1804.03492.pdf> [↑](#footnote-ref-17)
17. <https://github.com/mikacuy/pointnetvlad> [↑](#footnote-ref-18)
18. <https://www.tensorflow.org/api_docs/python/tf> [↑](#footnote-ref-19)
19. <https://arxiv.org/pdf/1804.03492.pdf> [↑](#footnote-ref-20)
20. <http://skm.kmi.open.ac.uk/cso/> [↑](#footnote-ref-21)
21. Examples include: tf.constant, tf.placeholder, tf.nn, tf.nn.conv2d, tf.nn.relu, tf.nn.tanh, tf.nn.softmax, etc. [↑](#footnote-ref-22)
22. Examples include: tf.Graph, tf.Tensor, tf.Module, tv.Variable, etc. [↑](#footnote-ref-23)
23. For example, the KGs constructed by the text2graph, image2graph and code2graph modules. [↑](#footnote-ref-24)
24. 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-25)
25. I. Goodfellow and Y. Bengio and A. Courville, “Deep Learning”, MIT Press, 2016 [↑](#footnote-ref-26)
26. 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-27)
27. T.N. Kipf and M. Welling, “Variational graph auto-encoders”, <https://arxiv.org/abs/1611.07308> [↑](#footnote-ref-28)
28. Yu, Shih Yuan, et al. "Pykg2vec: A Python Library for Knowledge Graph Embedding."; <https://arxiv.org/abs/1906.04239> [↑](#footnote-ref-29)
29. 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-30)
30. Those with titles “*DeLiGAN: Generative Adversarial Networks for Diverse and Limited Data*” <https://arxiv.org/abs/1706.02071> and “*Attentive Generative Adversarial Network for Raindrop removal from a single image*” <https://arxiv.org/abs/1711.10098> [↑](#footnote-ref-31)