**DARPA ASKE DCC – Milestone 9, 2019 (Month 13)**

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

**Contents**

[1. Introduction 2](#_Toc26179764)

[2. Text2Graph 2](#_Toc26179765)

[3. Image2Graph 3](#_Toc26179766)

[4. Code2Graph 10](#_Toc26179767)

# Introduction

The purpose of this report is to provide an overview of the work that has been done during **November 1st** to **November 30th, 2019**. The team has focused on comparisons of several methods in all modalities and provides initial results.

# Text2Graph

We have focused on enhancing the dataset of scientific papers describing DL models, architectures and algorithms. The enhanced dataset plays an important role in our evaluation and comparison of the developed named entity recognition (NER) and relation extraction (RE) statistical models. Both of these models need high quality of labeled data. This means text that has been annotated with entities and relations that are relevant and have a unique meaning in the area of Deep Learning. For example, a “*Convolutional Neural Network*” will be marked as a “*Method*”, whereas “*image processing*” or “*machine translation*” will be marked as “*Tasks*”. Similarly, the “*MNIST dataset*” will be marked as a “*Material*” needed to perform a specific statistical learning task, while the term “*accuracy*” or “*recall*” will be marked as “*Evaluation Metric*”. The process of annotating text to produce training datasets for NLP tasks is a time-consuming process and requires the participation of human experts who are hard to find and expensive to employ. In our case we have undertaken that task and used the web-based annotation tool Brat[[1]](#footnote-2) to create manual annotations. These annotations have formed our base dataset. It consists of 157 paper abstracts that describe Deep Learning models and architectures. These papers are taken from the website [PapersWithCode.com](https://paperswithcode.com). Most of these papers have been published in major scientific Machine Learning conferences (e.g., ICML, NeuriIPS, CVPR, etc).

To increase our labeled data (annotated text) we have developed a new process that can *automatically annotate DL papers*. This was achieved by combining (i.) our original annotations on the 157 papers, obtained using Brat, (ii.) external annotations provided by the sciIE project[[2]](#footnote-3), and (iii.) the Computer Science Ontology[[3]](#footnote-4), which is a large-scale ontology, covering the entire computer science. As a result, the labeled set has significantly increased. The total number of entities and relations that we have available are 5,539 and 4,647, respectively. Using these entities and relations within our automatic annotation tool we are now able to annotate full papers, not only the abstracts, in a shorter period of time. In our experiments so far we use the abstract and the introduction of the collected papers.

Our system uses a bidirectional LSTM (BiLSTM) along with a neural attention mechanism. We use an one-layer BiLSTM with 128 RNN units. In addition, we use an LSTM variant that introduces weighted peephole connections from the Constant Error Carousel to the gates of the same memory block[[4]](#footnote-5). The dropout parameters for the LSTM, feedforward networks and embeddings are: 0.3, 0.35 and 0.45, respectively. The input layer has dimensionality of 50 and the embedding dimension is 55. The attention mechanism is introduced in order to improve the focus on words that have *decisive effect* on the classification of relationship. We also use the sparse categorical entropy as the loss function. From our computational experiments we have realized that the attention mechanism can help increase the performance of our RE module.

We use as input the 157 papers, and our model is able to capture the essence of the scientific information that we intended to capture.

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Precision | Recall | F1 score |
| Text2graph-NER | 64.4 | 69.9 | 64.9 |
| Text2graph-RE | 77.1 | 68.6 | 71.0 |

**Table 1:** performance of the NER and RE implemented within the text2graph module.

As we can see from the above tables, our entity and relation extraction models are comparable with the state-of-the-art NER and RE methods (see for example the *LSTM-CRF*[[5]](#footnote-6) method which is used extensively for NER and the model *E2E-rels*[[6]](#footnote-7) *used for* relation extraction). Encouraged by the above results we plan to improve further by extending our dataset, adding more papers and annotate them with our automatic annotation system. This will enhance the performance of our statistical models and improve the quality of triples which are used for the construction of the knowledge graph generated by text. In addition, currently the tasks of NER and RE are performed separately (first we identify the entities and then the relations). We plan to follow a multitask approach where the entities and relations will be learned simultaneously.

# Image2Graph

In this work, we present a novel end-to-end framework that automatically localizes all figures from a research paper, classifies them, extracts the content of the DL architecture figures and represent it in the form of a graph. Given the PDF of a deep learning research paper, the image2graph module consists of four major steps: (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 milestone, we report performance of each individual module on our test dataset of 150 research papers in the next sections.

**1. Extract figures from research paper:**

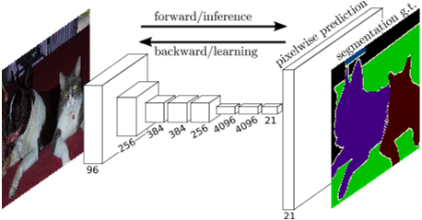
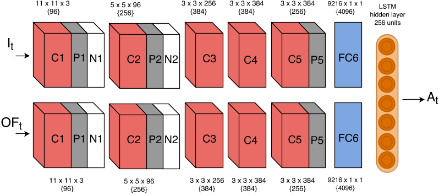
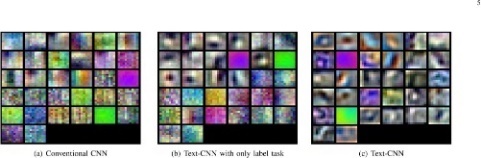
We used PDFFigures 2.04 tool for automatically extracting a list of figures from a research paper. It is reported to achieve 94% precision at 90% recall[[7]](#footnote-8).

2. **Extract figures from research paper:** Figure classification is performed in two steps. First, a binary classifier is used to identify the DL diagrams from all the figures extracted from a research paper. Accuracy of the two different binary classifiers pretrained using VGG16 and VGG19 is reported in Table 2.

It can be observed that the DL diagrams can be identified with 95.5% accuracy on the test dataset which is quite better than the 86.2% accuracy reported in dlpaper2code[[8]](#footnote-9). Figure 2 shows identification results for both correct and incorrect cases. It appeared the model predicted “other” to most of the incorrect “architecture” images which are the dominant classes in the provided training set. This problem can be overcome by adding more data for the other classes, either via data augmentation or by collecting figure data from more papers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1-score** | **Accuracy** |
| **Pretrained-VGG16** | 0.86 | 0.86 | 0.86 | 0.95 |
| **Pretrained-VGG19** | 0.86 | 0.86 | 0.86 | 0.94 |

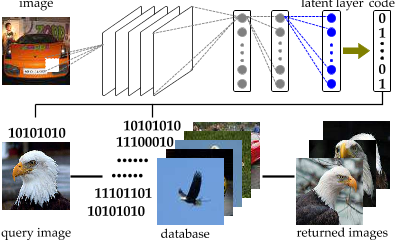
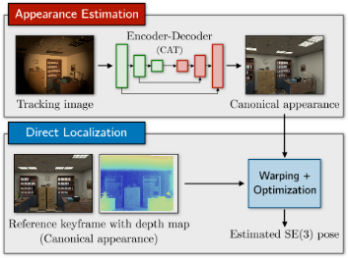
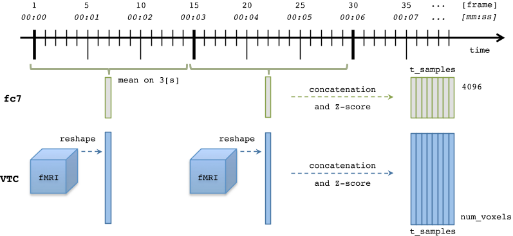
**Table 2:** The performance of two binary classifiers

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

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

**(a)**

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

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

**(b)**

**Figure 2:** Binary classification results: (a) Randomly sampled correct labels, (b) Randomly sampled incorrect labels

Next, we apply multi-class classifier to classify the DL diagrams into six classes: (a) Neurons plot, (b) 2D box, (c) Stacked2D box, (d) 3D box, (e) Matrix box, (f) Pipeline plot. The results are shown in Table 3. It can be observed that even on highly varying DL flow design images, F1 score is more than 70%.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1-score** | **Accuracy** |
| **Pretrained-VGG16** | 0.87 | 0.62 | 0.71 | 0.62 |
| **Pretrained-VGG19** | 0.87 | 0.60 | 0.70 | 0.60 |
| **Pretrained-Xception** | 0.85 | 0.60 | 0.71 | 0.60 |

**Table 3:** Performance of 3 multi-class classifiers

**3. Diagram analysis:**

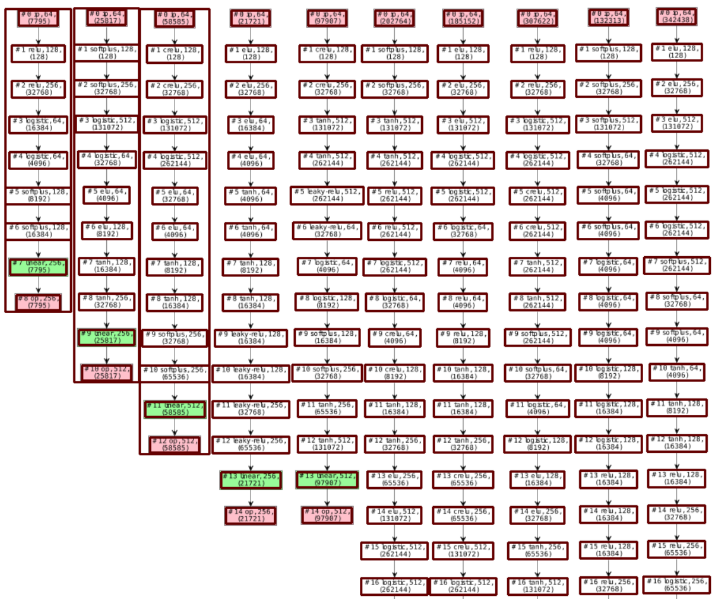
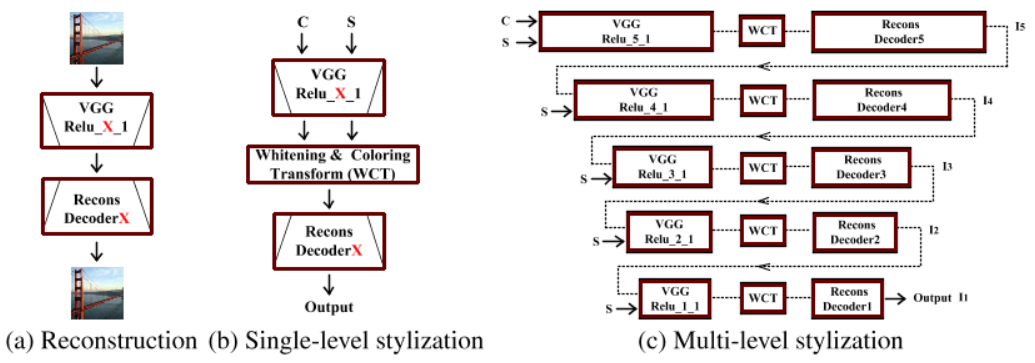
Once the figure to be analyzed further is determined, DL architecture analysis involves identifying the *nodes* and *text* components, followed by the edges connecting the nodes. Accuracy of each of these components are reported next.

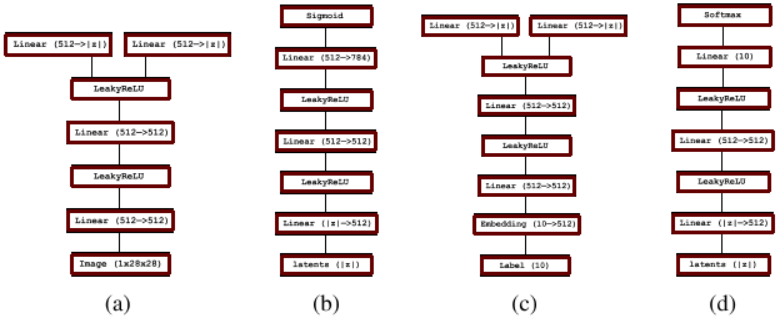
***Node detection:*** We apply a combination of contour detection with iterative region growing technique (which is especially useful when extracting touching or overlapping objects) to identify closed contours of the nodes. To evaluate the accuracy of the node detection algorithm we manually labeled all the components in test images of 150 research papers. The accuracy is measured using standard intersection-over-union (IOU) as follows:

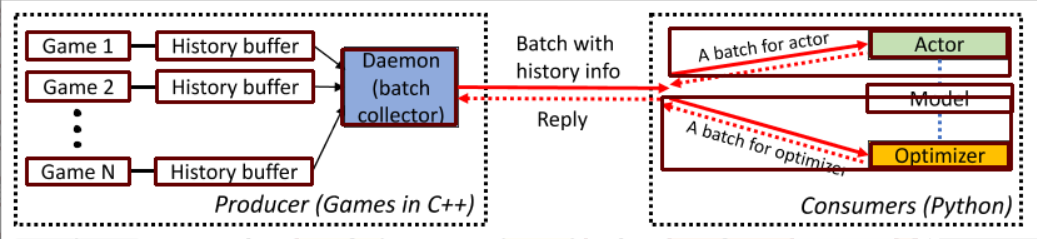
*Overall accuracy = w1\*(average of average IOU over all images) +*

*w2\* (accuracy of number of boxes detected)*

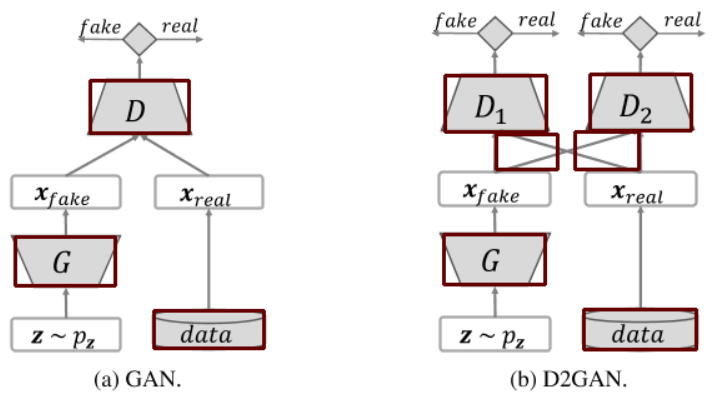
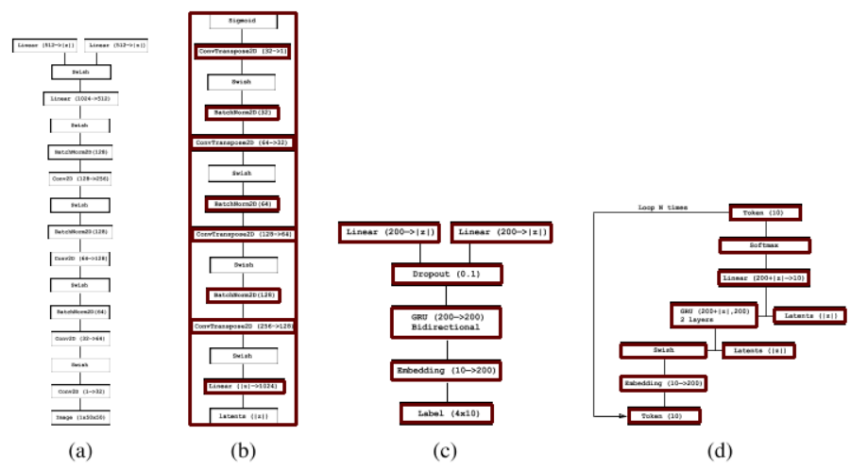
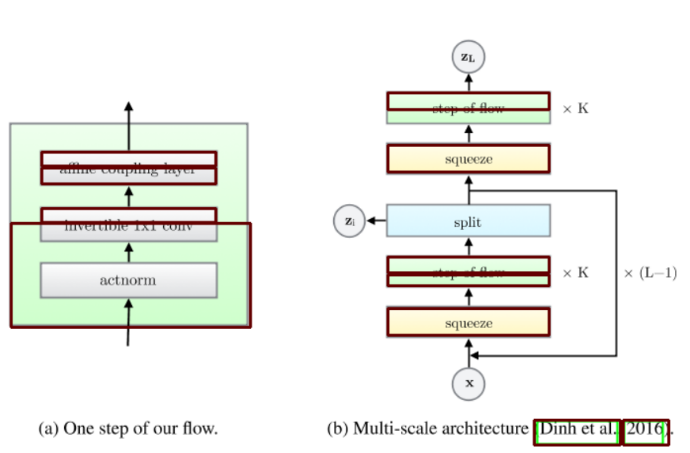
where w1 and w2 are weights set to 0.3 and 0.7, respectively. The node detection module achieved overall accuracy of 69.44%. Figure 3 and 4 show qualitative results of this module. In Figure 3 we show correct identifications of DL models/architectures and in Figure 4 we show bad or incomplete classifications. It can be observed that the node detection module was not performing well if the components are having irregular/fuzzy/dotted border, color-filled box with color transition. This is something that we will investigate in the remaining time of the project.







***Figure 3:*** *Example of good prediction of the node detection module*



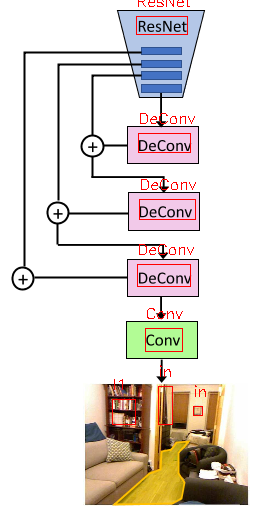
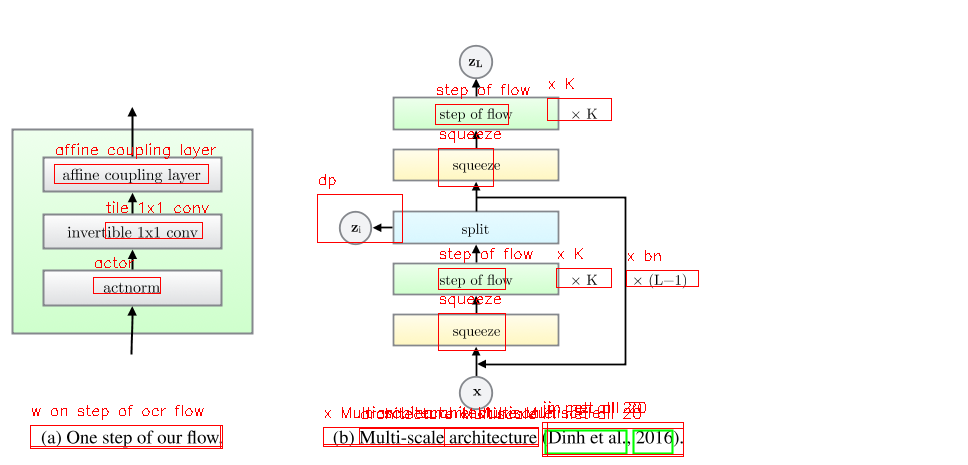
***Figure 4:*** *Example of bad/incomplete prediction of the node detection module*

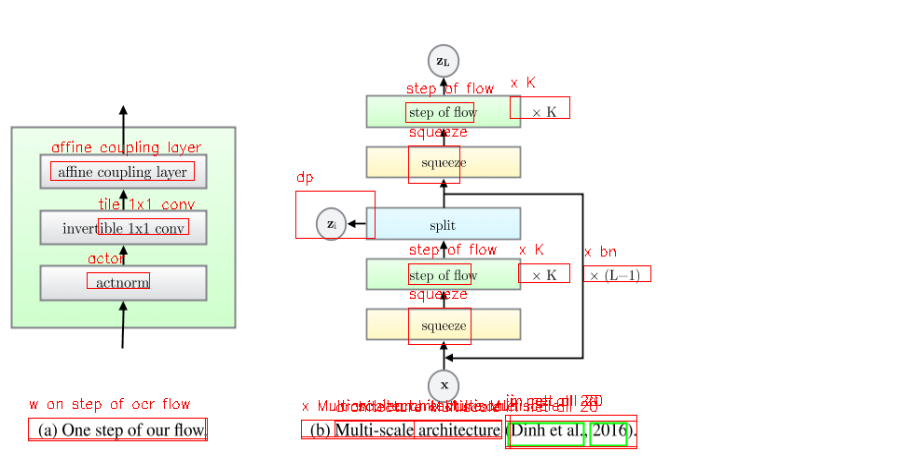
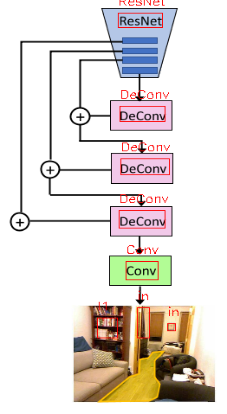
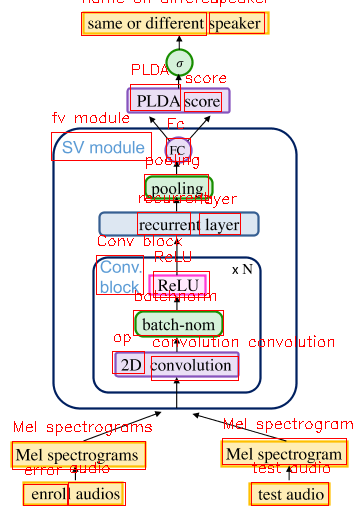
***Text detection:*** Once the nodes are extracted, the text in each image is obtained in two stages: *text detection* and *text recognition*.

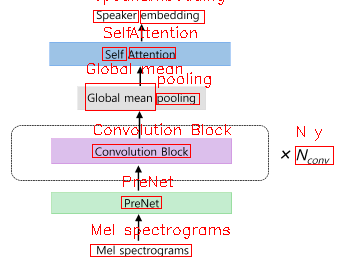
To evaluate text detection module, we manually labeled all the text boxes in the test images as before. Then text detection accuracy is measured using IOU as follows:

*Overall accuracy = w1\*(average of average IOU over all images) + w2\* (accuracy of number of text boxes detected)*

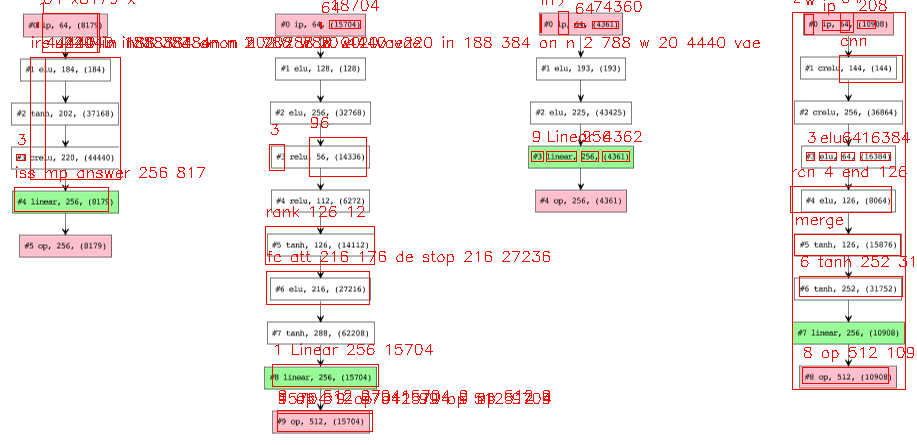
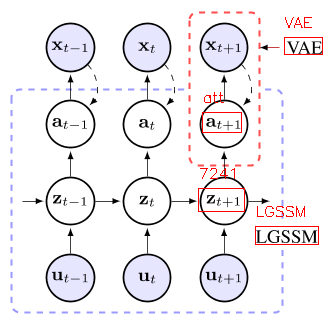
where are set to 0.3 and 0.7. Overall accuracy of text detection module was found to be 62.84%. Figures 5 and 6 show qualitative results of this module. It can be observed that the text detection module may not work well in challenging cases of low-resolution text, equations or symbols.







***Figure 5:*** *Example of good prediction of the text detection module*

***Figure 6:*** *Example of bad/incomplete prediction of the text detection module*

**4. Graph Construction:** After detecting DL design flow, a computational graph is created with following relationships describing each node or pair of nodes: Currently 10 predicates are used to represent a diagram as follows:

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

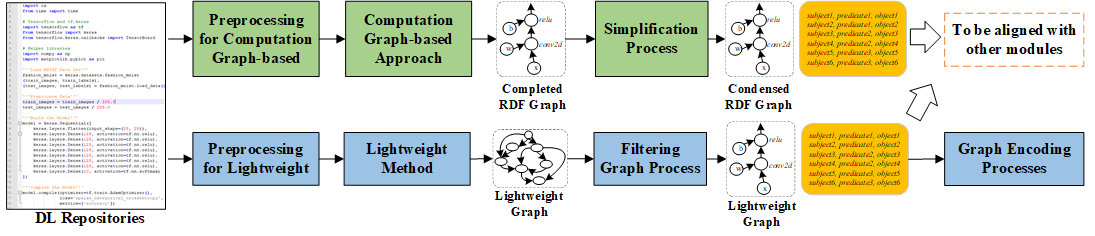
Our diagram-to-graph dataset contains 131 images with an average of 67 relationships per image. It depends on the complexity of the diagram image, the number of components present and their internal connectivity. A substantial fraction of the object annotations has overlapping bounding boxes and complex connectivity/relationships. The diagram-to-graph generation task is to simultaneously detect a set of components and predict the structural relationships between each pair of the detected components. A component is considered to be correctly detected if it has at least 0.5 IoU overlap with the ground-truth box. The accuracy metric measures the fraction of ground-truth relationship triplets <subject, predicate, object> that appear among the top 1 most confident triplet predictions in an image. The choice of this metric is due to the sparsity of the relationship annotations in Visual Genome — metrics like mAP would falsely penalize positive predictions on unlabeled relationships.

Accuracy of our model for diagram-to-graph generation task was found to be 59.86%. A substantial fraction of missed detection originates from lower accuracy of connectivity detection module and text detection module. For example, if the arrow heads are not detected properly, the “followedBy” relationship will not be detected accurately. Similarly, if the text parsing module does not generate correct output, “isType” or “hadDescription” predicates give wrong results. In our future work, we plan to focus on these two modules to improve overall diagram-to-graph generation accuracy.

# Code2Graph

For the code2graph module we have developed methodologies to extract the Resource Description Framework (RDF) representations from the code included in Deep Learning (DL) publications. We have developed two main approached: (a.) the Computational Graph-based Approach and (b.) the Lightweight Approach.

In the Computational Graph-based Approach (shown in green-colored boxes), 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. On the other hand, the Lightweight Approach (shown in blue-colored boxes) extracts the graphs by analyzing the abstract syntactic structure of the code itself. We have also add code-related definitions to a universal ontology shared across the entire DCC project. We then used this ontology to update the lightweight graphs generated from the code2graph module. The following sections describe the status of code2graph in MS 8. After MS 7, besides continuing the expansion of the datasets and updating the RDF formats for supergraph alignment, we have also formulated and applied several *graph embedding methods* for use in graph encoding processes to perform semantic inferences on DL repositories. Figure 7 shows the current code2graph pipeline architecture. The graphs generated from both approaches will be used in integrating with the supergraphs. Furhtermore, we will use the generated graphs, especially Lightweight graphs, to demonstrate methods that capture semantics from the source code.



**Figure 7. The current code2graph pipeline.**

**2. Tasks**

**2.1. Updating the Format of RDFs**

During Milestone 8, we changed the ontology file to “DeepSciKG.nt” in order to comply with the remaining modalities (text and images) in the project. In addition, we added the following metadata to the RDFs generated by the lightweight method:

code\_folder\_name om.type repository

code\_folder\_name isPartOfPublication input\_paper\_id

paper\_link\_id hasRepository code\_folder\_name

code\_folder\_name githubrepo github\_link

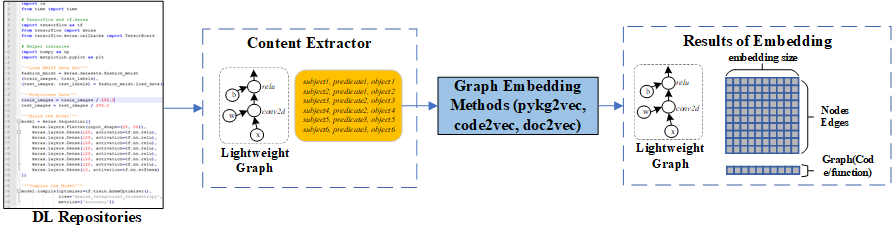
code\_folder\_name hasFiles file\_name

file\_name partOf code\_folder\_name

code\_folder\_name hasFunction function

These new entities and relation types have the same base namespace prefix: “<https://github.com/deepcurator/DCC/>”. The *code\_folder\_name* is the name of the directory that contains the source code and is usually the name of the GitHub repository. The *input\_paper\_id* is the paper identification tag input by the user. We have slightly modified the lightweight pipeline to allow users to specify *input\_paper\_id* and ontology files as input parameters. The *paper\_link\_id* is the filename of the downloaded ML paper, and *file\_name* is the name of python files. By adding this metadata, we increase the strength of the connection between graphs and their corresponding parent ML papers.

**2.2. Graph Embedding Methods**



**Figure 8. The architecture of performing code embedding task.**

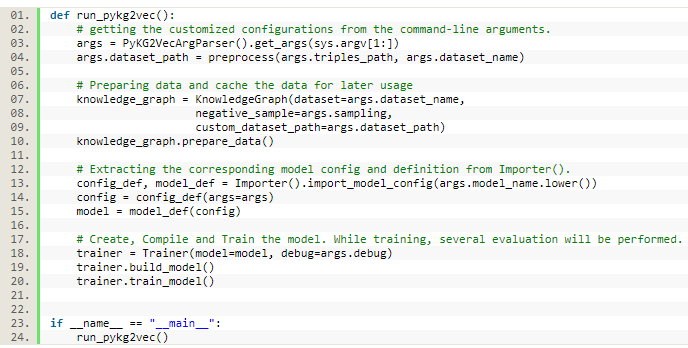
In order to find similar papers, determine originality, and perform code reconstruction, the code2graph analyzes the DL code associated with publications and extracts the information into an intermediate form that can capture the high-level semantics from DL publications. To do this, we have targeted to representing the source code or the code elements (e.g., functions defined by the developer of the source code) using low-dimensional latent vectors. The latent vector, also known as an embedding, captures the semantic properties of an object by representing it with its features distributed across multiple vector components. To learn the representations, we have explored the use of Knowledge Graph Embedding (KGE) techniques that capture the vectorized representations of relationships between nodes and neighbor nodes on the graph. Our own library: pykg2vec[[9]](#footnote-10) includes a collection of well known KGE models as well as tools for evaluating them. By leveraging pykg2vec, we are able to apply multiple translational-based and similarity-based KGE algorithms and compare the results for our datasets. Besides pykg2vec, we also explored the representation that captures the syntactic or structural information from the lightweight graphs[[10]](#footnote-11).

The KGE algorithm aims to find the representations for nodes and edges. In code2graph, a given lightweight graph is essentially a function call graph and we treated it as a knowledge graph, containing nodes that represent either internal function calls or TensorFlow function calls and edges that represent the calling or following relationships. For a given Knowledge Graph (KG), it contains a set of functions E and relations R between entities. The set of facts D+ in the Lightweight graph (extracted by using RDF’s N-Triple[[11]](#footnote-12) form) are then represented in the form of triples (h, r, t), where h, t ∈ E are referred to as the head and the tail functions and r ∈ R is referred to as the relationship (e.g., call or followed by). For example, the triple

“<Dense\_1> \t <followedBy> \t <Dense\_2>”

means the fact that the function Dense\_2 is executed right after Dense\_1. On the other hand, pykg2vec also implements the class Generator to create a set of unseen or negative training samples under the Open World Assumption (OWA). Then, we applied the pykg2vec library and trained different KGE models with different scoring functions fr(h,t)[[12]](#footnote-13). After training, the vectorized representations of the performance of embedding methods in pykg2vec is evaluated in terms of their capability of predicting the missing functions in negative triples (?, r, t) or (h, r, ?), or predicting whether an unseen fact is true or not. The evaluation metrics include the rank of the answer in the predicted list (mean rank), and the ratio of answers ranked top-k in the list (hit-k ratio), which will be shown in Table 4, Table 5 and Table 6.

In DCC repository, we have developed a script to tackle methods from pykg2vec as follows,



To trigger the script (script\_train\_pykg2vec.py[[13]](#footnote-14)), it requires the following command:

python script/script\_train\_pykg2vec.py -mn [model\_name] -tp [path\_to\_triples]

In script\_train\_pykg2vec.py, a stage of preprocessing is required before the training. First, all the triple information from the lightweight RDF graphs are combined altogether. Then, the data is shuffled and then splited into training, validation and testing sets with the proportion 6:2:2. From the command, the argument flag “-mn” allows the tester changing the models to to applied on the dataset, while “-tp” allows the tester to specify the root path storing the dataset.

In addition to learn the representations from KGE methods, an alternative way to acquire the vectorized representations for functions of a given Lightweight Graph is to utilize the syntactic structures extracted from the functions of the corresponding source code. In the pipeline of code2graph, we have also stored the Abstract Syntax Tree (AST) representations for each function node, which naturally containing more structural information in the form of parent-child node relationships. This additional information can be leveraged to improve our ability to capture the semantic and syntactic similarity of source code. One notable work which leverages ASTs for code embedding is code2vec. This technique decomposes a function-level source code AST into a collection of paths from its AST representation. This leads code2vec to model a given code snippet as a bag (multiset) of its extracted paths. During the learning process, code2vec learns a representation of each node on the AST as well as aggregated representations of sets of nodes.

The code2vec model was evaluated by using it to predict the function name for a given snippet of source code. Since Lightweight Graphs have an AST-like structure, we plan to implement a version of code2vec for Lightweight Graph embedding. This model will be evaluated against pykg2vec’s KGE models on the same metrics shown above. We believe that this will show a performance improvement due to the use of the additional structural information given by ASTs. This part will be listed as future steps.

**2.3 Results from Knowledge Graph Embedding Methods**

Since the format of RDFs produced by the lightweight method has been adjusted as described in Task 2.1, we also re-ran the lightweight method on our 60\_Lightweight dataset from MS 6. We selected and applied KGE methods from pykg2vec on reformatted datasets **60\_Lightweight, 60\_Lightweight with No Sequential Info,** and **170\_Lightweight**. For 60\_Lightweight No Sequential Info, we modified the lightweight method pipeline so that it removes all the related sequential information and indexing information from the names of entities. An example will be, from dataset **60\_Lightweight**, the entity with the name “**.testGraph\_extensive.Sequential\_2.Dense\_1**” will be changed to “**Dense**” in the 60\_Lightweight No Sequential. As for the 170\_Lightweight dataset, the PWCScraper has also continued collecting ML papers from paperswithcode.com. Currently, we have acquired 1560 ML papers in our database, 415 of which are TensorFlow papers. Of these 415 papers, we were able to successfully generate a Lightweight Graph for 170 TensorFlow papers.

We applied the same KGE methods on datasets **60\_Lightweight, 60\_Lightweight with No Sequential Info,** and **170\_Lightweight**. Each model in the KGE library was trained for 500 epochs using latent vectors with hidden\_size set to 100. The statistics of the dataset’s metadata information about the training and testing process are as shown in Table 4.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metadata Information** | **60\_Lightweight w/**  **No Sequential Info** | **60\_Lightweight** | **170\_Lightweight** |
| Total Training Triples | 36897 | 433367 | 477578 |
| Total Testing Triples | 4613 | 130085 | 143983 |
| Total validation Triples | 4613 | 86972 | 98353 |
| Total Entities | 4707 | 155400 | 172992 |
| Total Relations | 97 | 287 | 352 |

**Table 4. Statistics of 60\_Lightweight and 170\_Lightweight.**

The results from pykg2vec on the 60\_Lightweight and 60\_Lightweight with No Sequential Info datasets are shown in Table 5 and Table 6, while the results for 170\_Lightweight is shown in Table 7. From the comparison between 60\_Lightweight and 170\_Lightweight, we can see that the KGE methods provided by pykg2vec have consistent performance in terms of mean\_rank and hits10. Amid all the KGE methods, KG2E represents the entities and relations using multidimensional Gaussian distributions and it performs the best by having hit10 ratio 16.5% from the 60\_Lightweight dataset. Comparing training using TransE on both 60\_Lightweight and 60\_Lightweight with no sequential information datasets, the performance for the latter one performs much better than the former one. The reason is that in 60\_Lightweight, every function call is deliberately made unique by appending all the calling and following relationships extracted from its original source code. On the other hand, in 60\_Lightweight with no sequential information, most of the function calls are represented with its function names, making the models from pykg2vec learn the embeddings that captures the general semantic relationships between function calls. **From this, we learned that it is better to use the dataset without the sequential information***.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Models** | **Loss** | **mean\_rank** | **hits10** | **hits5** |
| TransH | 44.743 | 185.475 | 0.550 | 0.445 |
| TransE | 51.939 | 172.439 | 0.565 | 0.427 |

**Table 5. The results of KGE methods on 60\_Lightweight with no sequential information.**

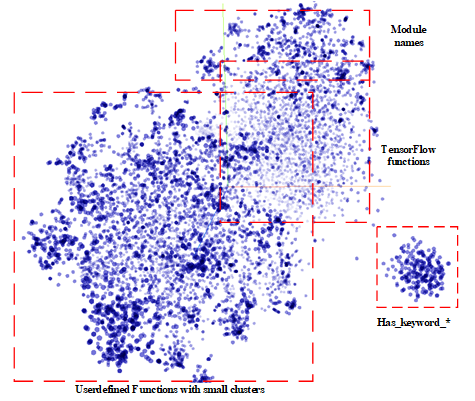
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Models** | **Loss** | **mean\_rank** | **hits10** | **hits5** |
| KG2E | 132.562 | 18905.424 | 0.161 | 0.119 |
| SLM | 1667.895 | 38410.438 | 0.081 | 0.063 |
| SME | 27535.370 | 72396.906 | 0.010 | 0.007 |
| TransD | 2410.188 | 13001.331 | 0.140 | 0.102 |
| TransE | 188.069 | 16590.602 | 0.100 | 0.074 |
| TransH | 181.402 | 16390.121 | 0.102 | 0.071 |
| TransM | 204.013 | 14340.157 | 0.124 | 0.091 |
| TransR | 118.556 | 78651.883 | 0.000 | 0.000 |

**Table 6. The results of KGE methods on 60\_Lightweight.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Models** | **Loss** | **mean\_rank** | **hits10** | **hits5** |
| NTN | 528.582 | 37510.890 | 0.018 | 0.014 |
| Rescal | 106454.627 | 36161.170 | 0.099 | 0.072 |
| RotatE | 1494.187 | 24964.170 | 0.041 | 0.032 |
| SLM | 1980.225 | 39988.930 | 0.066 | 0.052 |
| SME | 43050.780 | 83124.875 | 0.016 | 0.011 |
| TransD | 2845.462 | 12506.613 | 0.128 | 0.095 |
| TransE | 248.145 | 17008.910 | 0.102 | 0.073 |
| TransH | 193.713 | 17320.496 | 0.096 | 0.069 |
| TransM | 219.416 | 14168.570 | 0.116 | 0.092 |
| TransR | 172.482 | 86890.977 | 0.000 | 0.000 |

**Table 7. The results of KGE methods on 170\_Lightweight.**

The result appears consistent across all three datasets. To further demonstrate the results of applying pykg2vec, we use the T-SNE embedding visualizer as shown in Figure 9. The visualization of embeddings can demonstrate the distribution of all the entities (TensorFlow functions, Userdefined functions, modules) with techniques such as PCA and T-SNE. From Figure 9, 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 KGE methods alone, it is possible to capture the semantic meaning of source code and the entities on the Lightweight Graphs. However, as mentioned previously, significant performance improvements can be made by leveraging the structure of the code or functions. We discuss this future work in the following section.



**Figure 9. The visualization of all the function node embeddings with TransE.**

**3. Future Works**

**3.1 Expanding and Refining labelling information for datasets**

Looking further into the future, we believe that a combined KGE and code embedding approach can yield even better results. In addition, during the MS8, we have also found that fact that the amount and the quality of the dataset are still far from enough to perform significant inference tasks. Therefore, in the future what we will focus on 1) expanding the dataset of the Lightweight graphs and 2) refining the datasets by having better label information for source code and code elements.

**3.2 Enrich the semantic information on the embeddings captured by KGE methods**

The KGE methods from the pykg2vec library allow us to acquire the vectorized representations for both user defined functions and TensorFlow functions by capturing the calling relationships between those functions. However, our embedding approaches ignore the syntactic structures for user-defined functions into accounts, while such information can be easily acquired from the pipeline of the Lightweight Approach. An existing work has discussed the path-based approach for embedding functions using AST and has achieved a good performance on inferring the semantic names of functions. Therefore, we believe that we can further increase the level of semantics and improve the results of function embeddings by considering both the AST information for each user-defined function and the relationships between functions.

**3.3 Increase the abstraction level of embeddings**

In order to compare the similarities of publications, we have to increase the abstraction level for our embedding approaches to source code level or even project level. By applying whole graph embedding methods such as graph convolutional networks (GCNs) or Graph Auto-Encoders (GAEs), we expect the level of abstraction for the embeddings can be increased by taking both the node-attributes and the topological information in graphs into consideration.

# Knowledge Graph

Our effort has focused on two main areas: (a.) the design of our ontology which plays an important role in the creation of our knowledge graph and (b.) on the KG alignment

**Ontology Design**

We represent the extracted components as instances of a knowledge graph, which allows us to perform extraction, querying and visualization of the different modalities of the scientific paper and how they interact with one another. The schema of our knowledge graph DeepSciKG.nt sought to represent all the modalities in one single representation. This ontology development focused on utilizing domain expert created RDF and RDFS definitions spanning the modalities of – text, images, and code. 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:

1. The top level classes are:

CodeEntity, FigureComponent, Function, ImageComponent, Modality, Publication, PublicationAuthor, Repository, SourceCodeFile, TextEntity and tf

1. **CodeEntity** consists of two kinds of classes: TensorFlowDefined and UserDefined.
2. **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

1. **Function** and **ImageComponent** are used to categorize instances and map them to the right classes.
2. **Modality** consists of Code, Figure and Text, where Text is subdivided into AbstractText, BodyText, CaptionText and TitleText
3. **Publication** and **PublicationAuthor** focuses on representing just the publication metadata and author metadata
4. **Repository** and **SourceCodeFiles** are used for more fine-grained representations of source code data
5. **TextEntities** represent the set of classes that are extracted from the text2graph task. We currently extract – Method, Task, GenericTerm, Material, Evaluation Metric, and OtherScientificTerm
6. Finally, **tf** represents the hierarchy of modules that are present in the tensorflow documentation.

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 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 consolidated knowledge graph.

As stated above knowledge graph population focused on populating the schema with instances extracted from the different modalities. For this, we consolidated the papers from papers with code and organized them by conference type, year, and code framework. From the initial extraction of the dataset we were able to consolidate 1,848 papers. Out of these 1,848 papers we narrowed down on 739 papers which contain code repositories published in either tensorflow or pytorch. The datasets for both of these can be found in the repository development branches i.e. <https://github.com/deepcurator/DCC/tree/development/src>. The file pwc.csv shows the entire 1,848 papers, while the file pwc\_edited\_tensorflow\_pytorch.csv is the file that shows all the papers that we can process. The statistics for the knowledge graph are shown Table 8.

|  |  |
| --- | --- |
| **Modality** | **Graphs generated** |
| Text | 739 |
| Images | 151 |
| Code | 444 tensorflow graphs |

**Table 8:** Knowledge Graph construction statistics

**Graph Alignment:**

For graph alignment, we focus on two techniques. The techniques are summarized below:

***Implicit graph alignment:*** In this technique 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. For the first step in the demo we 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 were presented in the demo file and are present 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. The Computer Science ontology (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.

***Quantitative Evaluation of KG statements:*** For the knowledge graph evaluation process, in this phase we are currently collecting about 1,500 triple statements per modality. This would imply that we would evaluate a total of 4,500 statements. For each of these triple statements we will focus on extracting the most common entities i.e. text entity, image, entity and code entity. Then for these entities, we will print out all the relationship triples and the attribute value triples. Next, we would invite domain experts to manually label them based on their accuracy of capturing the knowledge, correctness of each component of the triple statement. This manual labeling will allow us to have a set of quantified ground truth values on which we will evaluate our extracted knowledge graphs in terms of precision and recall curves.

1. http://brat.nlplab.org [↑](#footnote-ref-2)
2. http://nlp.cs.washington.edu/sciIE/ [↑](#footnote-ref-3)
3. https://cso.kmi.open.ac.uk/home [↑](#footnote-ref-4)
4. https://arxiv.org/pdf/1503.04069.pdf [↑](#footnote-ref-5)
5. Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. In Proc. Conf. North American Assoc. for Computational Linguistics (NAACL). [↑](#footnote-ref-6)
6. Makoto Miwa and Mohit Bansal. 2016. End-to-end relation extraction using lstms on sequences and tree structures. In Proc. Annu. Meeting Assoc. for Computational Linguistics (ACL). pages 1105–1116. [↑](#footnote-ref-7)
7. Clark Christopher, and Santosh Divvala. "Pdffigures 2.0: Mining figures from research papers." In *2016 IEEE/ACM Joint Conference on Digital Libraries (JCDL)*, pp. 143-152. IEEE, 2016. <https://ai2-website.s3.amazonaws.com/publications/pdf2.0.pdf> [↑](#footnote-ref-8)
8. Sethi, Akshay, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. "DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers." arXiv preprint arXiv:1711.03543 (2017). [↑](#footnote-ref-9)
9. Yu, Shih Yuan, et al. "Pykg2vec: A Python Library for Knowledge Graph Embedding." *arXiv preprint arXiv:1906.04239* (2019). [↑](#footnote-ref-10)
10. He, Shizhu, et al., "Learning to represent knowledge graphs with gaussian embedding." Proceedings of the 24th ACM International on Conference on Information and Knowledge Management. ACM, 2015. [↑](#footnote-ref-11)
11. <https://www.w3.org/TR/n-triples/> [↑](#footnote-ref-12)
12. Refer to <https://github.com/Sujit-O/pykg2vec> for supported KGE methods. [↑](#footnote-ref-13)
13. Script: <https://github.com/deepcurator/DCC/blob/development/src/code2graph/script/emb/script_train_pykg2vec.py> [↑](#footnote-ref-14)