2018 25th Asia-Pacific Software Engineering Conference (APSEC)

DeepGraph: A PyCharm Tool for Visualizing and Understanding Deep Learning Models

Qiang Hu[1], Lei Ma[2], and Jianjun Zhao[1]
[1]Kyushu University, Japan [2]Harbin Institute of Technology, China

***Abstract*—As more and more domain specific big data become available, there comes a strong need on the fast development and deployment of deep learning (DL) systems with high quality for domain specific applications, including many safety-critical scenarios. In traditional software engineering, software visualization plays an important role in improving developers' performance with various available tools. However, there are limited visualization supports existing for DL systems, especially in integrated development environments (IDEs) that allow a developer to visualize the source code of a deep neural network (DNN) and its corresponding graph architecture. In this paper, we propose *DeepGraph*, a visualization tool for visualizing and understanding deep neural networks. *DeepGraph* analyzes the training program to construct the graph representation of a DNN, and establishes and maintains the linkage (mapping) between the source code of the training program and its corresponding neural network architecture. We implemented *DeepGraph* as a PyCharm plugin and performed preliminary empirical study to demonstrate its usefulness for understanding deep neural networks.**

***Index Terms*—Deep Learning, Visualization, Deep Neural Network, Code Mapping**

I. Introduction

In the past decades, deep learning (DL) experienced unprecedented rapid development and is successfully applied to many domain specific applications. Deep neural network (DNN) plays as the core component of DL-based software. However, a deep neural network is often complex in nature, with many layer components with diverse types and functionalities. Building a practical DNN often takes huge human efforts, and the development procedure is often error-prone. Currently, several DL frameworks such as PyTorch, TensorFlow, Keras have become available, which alleviate the development effort to some extent. Since the DNN is programmed in the form of traditional program (e.g., Java, Python), the developers may easily make mistakes without in depth thought and understanding of the target DNN to build. When the run time behavior of a training program goes beyond expectation, it is also difficult for a developer to localize where the issues hide. As a well established technique for traditional software, software visualization enables the facilitation of the understanding on the software under analysis. If the visualization of complicated DNN source code becomes available, it could be of great value for understanding and giving immediate feedback during DL development.

Although there is some preliminary work for DNN visualization, they treat the DNN programming and visualization as separate processes and do not give immediate mapping between the DNN source code and DNN architecture component for visualization. Therefore, it would be important to provide a tool which can support direct visualization as feedback during DNN development. Moreover, a synchronized code mapping between DNN source code and DNN model architecture visualization could potentially boost the DNN development cycle. In traditional software engineering, visualization and code mapping are generally both integrated into one supporting tool, however, to the best of our knowledge, such tools which include both functions are still not available for deep neural networks.

In this paper, we present *DeepGraph*, an automated visualization and code mapping tool for constructing the data flow graph representation of the architecture from DNN source code and automatically synchronize between source code and its graph representation. *DeepGraph* is implemented as a PyCharm plugin for supporting TensorFlow, the current most widely used DL framework. *DeepGraph* uses PyCharm embedded window to display the model's graph, and users can find relevant code corresponding to the node by using shortcuts after selected one node. The main contributions of this paper can be summarized as follows:

- We proposed a visualization technique of DNN which provides a novel DL software development process. It constructs DNN graph representation from source code for visualization, establishes and maintains the code mapping.
- We solved two key issues to enable *DeepGraph* to handle practical DNNs: a) Use a simple and effective web structure to overcome the problem that a tool window in PyCharm has low performance and it can not be graphically displayed with a high-performance browser like TensorBoard [1]. b) Use keyword search technology to do code mapping while modeling code for neural networks is different from the traditional code.
- We implemented the proposed visualization technique in *DeepGraph* as a plugin for PyCharm and showed the effectiveness of it for understanding deep neural networks through a preliminary empirical study.

The rest of this paper is organized as follows. Section II compares *DeepGraph* with related work. Section III describes the design, implementation and preliminary empirical results of *DeepGraph*. Section IV concludes and discusses the future work of *DeepGraph*.

978-1-7281-1970-0/18/$31.00 ©2018 IEEE
DOI 10.1109/APSEC.2018.00079

II. Related Work

In this section, we attempt to review the most relevant work in three aspects: visualization of traditional software, visualization of DL models, and code mapping.

A. Visualization of Traditional Software

In traditional software, visualization tools assist developers to comprehend, debug, which could also generate code automatically. The widely used GUI plugin (e.g., Eclipse SWT-Designer [2]) provides visual code generation capabilities, and it supports developers to drag the components to build their applications. However, SWT-Designer focuses on the modular code generation and it doesn't contain the programming graph visualization which is the biggest difference from *DeepGraph*. There are also some PyCharm tools for visualzation, SVN Revision Graph [3] visualizes the SVN version controller and Graph Database Support [4] visualizes the database management. Alsallakh et al. present a visualization tool [5] to help debug the program that changes the data values to bar charts, histograms and line chart. Compared with these three tools, *DeepGraph* visualizes the data flow graph instead of data values and makes DNN models more intuitive.

B. Visualization of DL Models

Visualizing the neural network model is already an active research topic during the last several years. There are some tools provide graph visualization capabilities such as TensorBoard, which is a part of the TensorFlow machine intelligence platform. And the Netscope [6], an online visual tool that supports the neural network structure described by the prototxt format, is also a tool for DL visualization. However, both of them visualize by using another platform after the model is built while *DeepGraph* dose not need to use another platform to visualize. There are some other works for visualizing neural networks [7], [8], which target to visualize in specific directions and do not present the workflow of the model. Recently, the Neural Network Console [9] released by Sony uses visual technology to build, train and test models, but it's completely out of the code. For understanding DL models, Guo et al. make a review of various DL approaches and their recent developments [10]. What differs *DeepGraph* from these works is that *DeepGraph* aims to combine both code and graphics to help developers to understand DL models.

C. Code Mapping

Code mapping function needs to analyze source code. There are some methods to analyze code, the basic control flow analysis [11] presented by Allen is widely used, it checks the branches and loops of the program and then gives information about all possible paths and dependencies between the statements, and also expends to higher-order languages. Another widely used method is data flow analysis [12] which collects the semantic information of the program from the program code and determines the definition and use of variables at compile time through algebraic methods. These analysis methods work for the entire structure of the code but *DeepGraph* uses indentation of Python and TensorFlow's keywords for code analysis to reduce the massive calculations.

III. *DeepGraph*: A PyCharm Tool For Visualizing and Understanding DL Models

We next present the design rational, the implementation issues, and the preliminary empirical results regarding *DeepGraph*.

A. Design

1) Workflow and Layout: For the development of DL systems, although each team has its own development process, generally, the basic work flow is like this: design, prepare data, program, in most cases, after analyzing the result, adjust hyperparameters and train the model again, then compare the results and choose the best hyperparameters combinations for the model. Generally, developers design the neural networks by their experiences and draw the layers and data graph of the network at the start of the process. In order to check their models, they usually observe the training and test results at the end of the process. Furthermore, developers may print each layer's data structure to judge if their models are right or not. Developers have to use other tools to visualize the model if it is necessary. When PyCharm and Tensorflow are used to do this work, the traditional work flow is shown in "Fig. 1". We can find that three processes (coding,build and run, result in the console and analysis) are in PyCharm, while the visualizing process is out of it. This is obvious inconvenient for developers.

In this work, we aim to make the entire development process easier, so making the visualization process be in the same tool with the coding and other processes is a task of *DeepGraph*. To achieve it, we add *DeepGraph* as a visualization plugin

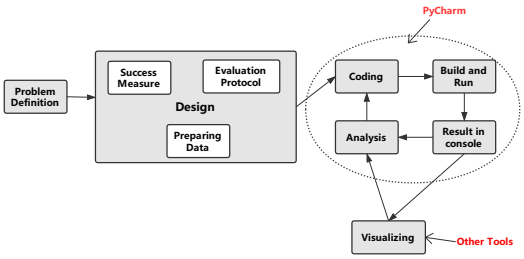

Fig. 1. Traditional work flow of DL development in PyCharm.

into PyCharm. *DeepGraph* visualizes the graph at the same time of the training period which is depicted in "Fig. 2". After training the model, the model structure is displayed in the PyCharm tool window and developers can observe them directly. To better understand the connection between models and their code, find code blocks quickly, *DeepGraph*'s further work is to inversely correlate graph and code. To do so, a corresponding relationship between code and a mouse listener

are added to each node of the visualization graph for listening users' actions. When the mouse cursor over the node and the user presses the shortcut, *DeepGraph* will highlight the code it related.

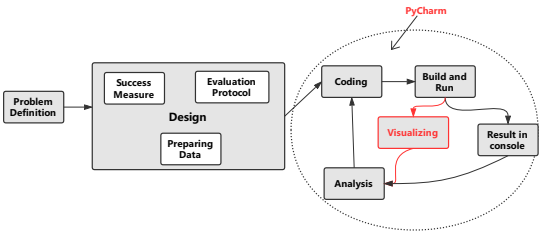

Fig. 2. The workflow of DL development in PyCharm used *DeepGraph*.

Functionality, performance and convenience are all the most important metrics of a plugin and they are what we most concerned about. We add *DeepGraph*'s window to the toolbar on the right side of PyCharm like the SciView. In addition, we add a tool button (in View - Tool Windows) to control the activation/inactivation. When *DeepGraph* is activated, the graphic display window will occupy the right side of the code edit bar, and users can freely adjust the size and position of this window. The graph drawn by *DeepGraph* also displays the model hierarchically like the other tools. In the initial state, *DeepGraph* just draws the overall neural network hierarchy(e.g., *layer1*, *input*), and the layout can be updated by users. For performance reasons, we only show the most important information(e.g., nodes, edges) on the graph. About the node-to-code mapping function, we use PyCharm's code highlight technology to showcase the results from the code analysis.

DeepGraph has two main functions: visualization and code mapping, the key point of the implementation of these two parts is basically different.

2) Visualization: Although GUI programming can do this work, in order to have better scalability, we draw the graphics onto the web page. However, finding a web engine that can be embedded in PyCharm is a challenge that must be overcome. Another challenge is that a multi-level nested web structure is hard to use in building the web because of the performance limitations of integrated environment. We intend to use the least page hierarchy to complete this work. We visualize TensorFlow graphs thought the follow steps:

Get graph data. To use the *DeepGraph*, users need to call *Tensorflow* APIs in their code to save *summaries* to log files. At the training process, the models' event data (data flow information, training and validation accuracy of each epochs et al,) will be saved in the log files. When *DeepGraph* runs, it searches and loads the log files, then gets the data flow information as graph data for the next step.

Construct the graphical structure. We extract two arrays *node* and *edge* from the data structure that we get in the previous step. Then follow these steps to build our data structures:

a) Convert the *node* whose data structure is $tf.node$ to a normal array $Nodes$ and set its structure as $[nodeName, groupName]$. The first data in $Nodes$ is the node's name and the second one is the node's group. And so as the *edge*, convert it as $Edges[nodeA, nodeB]$.

b) Each $tf.node$ has a direct $parent$ attribute and we extract it as a group data. Create a new array $Groups$ to save the group data and set its data sructure as $[groupA, groupB]$. Then initialize $Groups$ as $[group]$.

c) The $Nodes$ and $Edges$ are used for searching but not for drawing due to that they contain all the data we need which can not be changed. Therefore, we create a new array $drawEdges$ to save the edges used for drawing, and in fact the $Groups$ stored content is the same as $Nodes$'s, the reason is if one node has a child, it can be treated as a group, so we use $Groups$'s data for drawing.

Initial and update drawing data. In the $tf.node$, if the nodes belong to the highest level, they don't have parent and their parent attributes are set as $\_root\_$. In our work, we change it as $group$ and set it as the initial data of $Groups$. We initialize and update the $Groups$ and $drawEdges$ through following steps:

a) Find the nodes. Traverse the $Nodes$ and $Groups$, for each of $Groups$, search all the nodes and find that if $Node_{nodeB} == groupA$, then add it to the $Groups$ as $[Node_{nodeA}, groupA]$.

b) Find the edges. The array $Edges$ contains all the node-to-node edges, but there are only one edge to connect two nodes in our design. One node may have children, so the data in $Edges$ can not be used directly. Find the edges which connect two groups in the array *Groups*, then add them to the *drawEdges* which are used as the drawing data, and we use the Algorithm 1 to do it.

Algorithm 1 findEdges

```
TemGroups ← Groups;
Delete groups: groups ∈ TemGroups &&
               groups_parent ∈ TemGroups;
GroupNum ← number of TemGroups;
for edge in Edges do
  for i = 0 to GroupNum do
    for j = i + 1 to GroupNum do
      if TemGroups[i][0] ∈ edge
      && TemGroups[j][0] ∈ edge then
        Add edge to drawEdges;
      else
        Iterate over the children of
        TemGroups[i][0] and TemGroups[j][0],
        if find, break;
      end if
    end for
  end for
end for
```

c) When users double click on a node, *DeepGraph* will search the information of it from $Nodes$ and add $[Node_{clicked}, Group_{clicked}]$ to the $Groups$, and then repeat steps a through b.

d) When users double click on a group, *DeepGraph* will delete all the groups in the $Groups$ which $Group_{groupB} == Group_{clicked}$, and then repeat steps a through b.

e) Use $Groups$ and $drawEdges$ data as the drawing data to visualize the graph, when data changed, refresh all the graphs to do re-drawing.

3) Code Mapping: Code mapping is a complicated work. In most cases, abstract syntax trees are used for source code analysis which need to convert source code to a tree diagram, and each node on the tree represents a structure in the source code. But in this work, we target the Python version of TensorFlow for node-to-code mapping. Therefore, the core thing we need to know is the part of code that generates the node. In general, the code in this part doesn't have many conditional branches, so we can analyze it at the code level.

When TensorFlow is used to do the DL development, the source code can be roughly divided into three parts. The first part uses TensorFlow's API to build the model(e.g., number of layers, structure of each layer), defines the data type of the input and output. The second part loads the data from files, processes the data then converts it to the data type and structure of the model's input. And the third part sets hyperparameters (e.g., epochs, batch_size) for model training and then trains the model. Since *DeepGraph* only visualizes the data flow for the DL model, the nodes for the process of data preprocessing are unnecessary to be considered. The program mode of DL, where one model needs massive data for training, is different from the traditional one. As it's difficult to track changes and flows of each data, we just care about the data structure of each node. It means in our mapping work, it's unnecessary to include logic code of the second part, we can selectively skip it. In fact, *DeepGraph*'s mapping function only covers the first part of code and all the variables.

In order to avoid massive calculations in the interaction process, we construct the correspondence between nodes and code immediately when the plugin runs. *DeepGraph*'s code mapping works as follows: First of all, it removes all comments from the source code, since we need to locate the exact location of the code, it replaces the text in the comment with a character 'T'. Then it uses Python's indentation feature to block the code and divide all the code by scope, saves it as map. Then it executes a simple algorithm as shown in Algorithm 2 to get the map that stores all possible node names and corresponding code intervals. Finally, when the front end of the graphic transmits the node name, the back end of plugin searches directly in this map and then highlights the code.

Algorithm 2 BUILD_RELATIONSHIP($Code$, $lineNum$)

Initialize
KeyWordMaps[keyword, position],
Functions[name, parameters], *Variables[name, value]*,
FunctionObjs[name, parameters], *TFKeyWords[]*;
for $i = 0$ to *lineNum* **do**
  if *function defines* or *variable defines* **then**
    update *Functions* or *Variables*;
  end if
end for
Processing Variables, only retains variables and real values(e.g., [A, "a"]);
for $i = 0$ to *lineNum* **do**
  Find the function call which $\in Functions$, *add it to FunctionObjs*;
end for
for key in *KeyWordMaps* **do**
  for $i = 0$ to *lineNum* **do**
    if $key \in Code[i]$ **then**
      if *Code[i]* in *function part* **then**
        find the function call in FunctionObjs, add the name in FunctionObjs and position to KeyWordMaps;
      else
        add the name in this key word part and position to KeyWordMaps;
      end if
    end if
  end for
end for
return $KeyWordMaps$;

B. Implementation

We implemented *DeepGraph* as a PyCharm plugin and published it as an open-source[1]. To get the graph information, we use the backend of TensorBoard to parse the log files generated by the model. Since the IntelliJ plugin development can only use Java language, to run the Python code, *DeepGraph* uses *com.intellij.execution.RunManagerEx* to call the program execution interface to run the back end Python code first to get the data by using the configuration that the users' environment used. We also use Intellij's OpenAPI to build the plugin and JavaFX's WebEngine as a platform for drawing graphics. We create a drawing board on the web page by using D3.js and use Dagre-d3 to implement automatic layout of nodes and edges which acts as a front-end to *dagre*, providing actual rendering using D3. And we update the layout using the methods mentioned in the previous *Visualization* section.

At the beginning of the plugin's running, we use the Algorithm 2 to generate a map. Then we use the *netscape.javascript.JSObject* library and the function *setMember* to implement mutual calls between Java and JavaScript. Moreover, code highlight function implemented via feedback from the graphical listener and using the methods defined in *com.intellij.codeInsight.highlighting.HighlightManager*.

[1]https://github.com/wellido/DeepGraph

C. Preliminary experimental results

We use example code in TensorFlow's GitHub as our source code to test *DeepGraph* at this stage. Firstly, we run the code to build a model and generate log files. Secondly, we activate *DeepGraph* by using shortcut or clicking button in the tool bar as shown in Fig.3. Thirdly, we use shortcut to activate *DeepGraphView*. One of the visualization results are shown in Fig.4. Finally, we test the code mapping function. Currently we have added two TensorFlow keywords *tf.name_scope* and *tf.placeholder* to our node-to-code mapping algorithm. After selecting a node by mouse cursor, and using shortcut to activate the mapping action, the corresponding code is highlighted as shown in Fig.5. We can also verify our visualization results by the code mapping function.

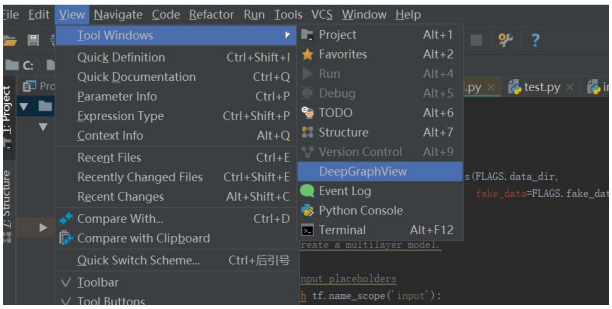

Fig. 3. *DeepGraph* registration location.

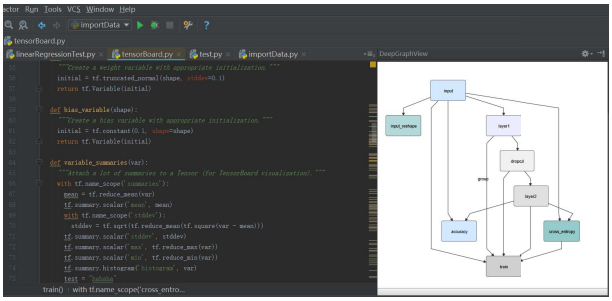

Fig. 4. DeepGraphView.

Preliminary work of *DeepGraph* has achieved unified DL development platform. Now it's unnecessary to use a second platform to visualize the model but just use PyCharm. And the node-to-code function allows developers to get feedback from graphics which is not available in other visualization tools. All the information makes us believe that it is necessary to carry out further development.

IV. Conclusion and Future Work

In this paper, we have presented *DeepGraph*, a PyCharm tool which combines visualization and code mapping functions for the Python version of TensorFlow. Benefiting from combining coding with visual analysis, *DeepGraph* reduces the time developers waste in tool conversions. And we also present a simple node-to-code algorithm that is easy to extend.

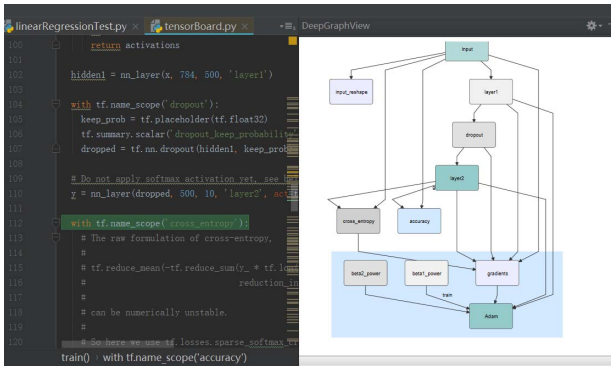

Fig. 5. Node-to-Code mapping in *DeepGraph*.

In the future, we would like to solve the limitations of current *DeepGraph*. We would like to solve the problem that the path of log files is specified and will add all the TensorFlow's keywords for comprehensive code mapping. We also consider to extract the model structure by using one training data before drawing the graph in order to save the time for training.

Acknowledgment

This work was partially supported by 973 Program in China (No. 2015CB352203) and JSPS KAKENHI Grant 18H04097.

References

[1] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 1–12, 2018.

[2] D. Rubel, J. Wren, and E. Clayberg, *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011.

[3] D. Batrak, "Svn revision graph," https://plugins.jetbrains.com/plugin/6178-svn-revision-graph, accessed Nov 07, 2014.

[4] V. Dmitry and K. Sergey, "Graph database support," https://plugins.jetbrains.com/plugin/8087-graph-database-support, accessed Jun 29, 2018.

[5] B. Alsallakh, P. Bodesinsky, S. Miksch, and D. Nasseri, "Visualizing arrays in the eclipse java ide," in *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 541–544.

[6] Netscope, "Netscope-neural network visualizer," http://ethereon.github.io/netscope/quickstart.html, accessed July 19, 2017.

[7] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*, 2015.

[8] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[9] Sony, "Neural network console," https://dl.sony.com//.

[10] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016.

[11] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[12] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.