DeepVisual: A Visual Programming Tool for Deep Learning Systems

Chao Xie, Hua Qi, Lei Ma, and Jianjun Zhao
Kyushu University, Japan
{xie.chao.792, qi.hua.677}@s.kyushu-u.ac.jp, {malei, zhao}@ait.kyushu-u.ac.jp

***Abstract*—As deep learning (DL) opens the way to many technological innovations in a wild range of fields, more and more researchers and developers from diverse domains start to take advantage of DLs. In many circumstances, a developer leverages a DL framework and programs the training software in the form of source code (*e.g.*, Python, Java). However, not all of the developers across domains are skilled at programming. It is highly desirable to provide a way so that a developer could focus on how to design and optimize their DL systems instead of spending too much time on programming. To simplify the programming process towards saving time and effort especially for beginners, we propose and implement *DeepVisual*, a visual programming tool for the design and development of DL systems. *DeepVisual* represents each layer of a neural network as a component. A user can drag-and-drop components to design and build a DL model, after which the training code is automatically generated. Moreover, *DeepVisual* supports to extract the neural network architecture on the given source code as input. We implement *DeepVisual* as a PyCharm plugin and demonstrate its usefulness on two typical use cases.**

***Index Terms*—Deep Learning, Deep Neural Networks, Visual Programming, Visualization**

I. Introduction

Recently, deep learning (DL) experienced rapid progress and achieved competitive performance in numerous areas such as image recognition, natural language processing, autonomous driving, medical diagnose and drug discovery. More and more new demands and requirements to adopt DL solutions from different domains started to emerge. Meanwhile, companies and organizations rush to develop DL frameworks for DL models development, aiming to dominate the infrastructure of DL software development. Up to present, quite a few DL frameworks are available such as Tensorflow [1], Keras [2], Caffe [3], and CNTK [4]. However, these frameworks often require a user to have a certain level of programming skill. Considering the users from diverse domains intend to take advantage of DL solutions, some of whom may have little programming background, a more accessible DL development technique is highly desirable.

In traditional software development, visual programming is an effective technique to simplify programming, which supports to manipulate program elements graphically. Towards assisting DL system developers to better understand the DL systems, some visualization techniques are proposed with tools available, such as TensorFlow PlayGround [5] that enables a user to observe how the roles of neurons are shaped during the training procedure and TensorBoard [6] that provides visualized feedback on the runtime training behavior to help developers optimize the DL system. However, they still do not provide visual programming support to ease the DL system development. While some commercial visual programming IDEs of DL start to emerge recently, there still lacks a publicly available and accessible open-source visual programming tool in the research community. For instance, *RapidMiner Studio* enables users to build a DL model visually, but it was designed for data science specialist at the very beginning, whose operation is rather complicated for a DL beginner. *Neural Network Console* developed by *Sony* provides entirely visual development of the DL system. However, the DL libraries (framework) it supports have quite limited user scope [7].

To bridge such a gap, we propose *DeepVisual*, a graphical user interface (GUI) programming tool for DL. *DeepVisual* visualizes the programming process of DL system development, from data preprocessing, designing neural network architecture, configuring hyperparameters, to generating training program code of a target DL system framework. As a significant programming element of the deep neural network (DNN), a layer is represented as a visualized component in *DeepVisual*. With *DeepVisual*, a DL software developer can drag-and-drop the layer components, connect them, and set hyper-parameters to design and create a neural network visually, after which the training program of a target DL framework is automatically generated, which is faster, more user-friendly, and of less chance to make mistakes compared to program source code line by line.

Moreover, *DeepVisual* also supports to automatically construct the structure of a DNN from the source code written by DL software developers. This function is especially useful for a DL beginner, who can import the unfamiliar training code and visualize its architecture design and configurations. *DeepVisual* establishes the bidirectional correspondence between the visual program and source-code program of a DL system, which potentially facilitates the understanding, debugging and further analysis of the DL software under development.

The contributions of this paper are summarized as follows:

- We proposed *DeepVisual*, a visual programming tool to assist the DL software development. It visualizes the whole programming process during DL development.
- We addressed the two critical issues of visual development in *DeepVisual*: 1) Generate the code for a target DL framework based on the structure graph constructed in *DeepVisual*, 2) Extract and reconstruct the neural network structure via importing the training program source code.

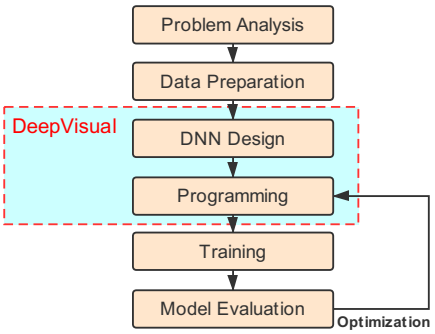


Fig. 1. The workflow of typical development process of deep learning, where the highlighted part illustrates the processes visualized by *DeepVisual*.

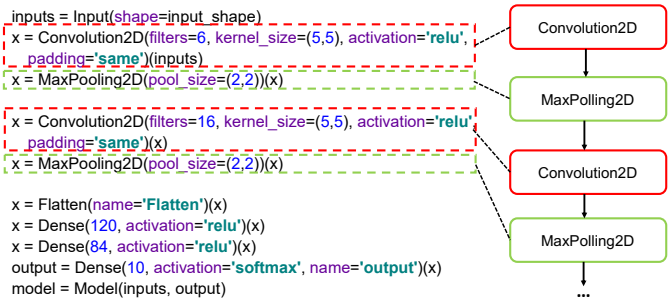


Fig. 2. An example of the graphic representation in DL framework Keras

- We implemented *DeepVisual* as a PyCharm plugin and demonstrated its usefulness via two typical use cases.

II. *DeepVisual*: A Visual Programming Tool for Deep Learning

A. The Workflow with DeepVisual

In a typical development process of DL (see Fig. 1), after analyzing the problem and preparing the data set, developers design the neural network and, like traditional software development, type the code line by line for tasks like data preprocessing and neural network construction. Then, the training process starts to optimize the DL model performance on a given task, which shapes the DL decision logic. Based on the feedback of the training process, the developer adjusts the network structure and fine-tunes hyper-parameters towards further performance enhancement. Such development steps often go through several iterations until a trained model meets the specified requirements.

In practice, during the whole cycle of DL development, the optimization and tuning of the neural network performance are much more critical, where a developer often spends more efforts on rather than programming. *DeepVisual* aims to reduce such manual effort by providing a DL developer with visual development of the DL software. In general, a DL developer designs and sketches the neural network on draft paper, based on which the DL system is implemented by programming. In *DeepVisual*, each layer of the neural network is represented as a visual component and the developer can drag-and-drop components to design and implement a neural network simultaneously. Then, *DeepVisual* dynamically generates the corresponding code in a targeted DL framework. Our *DeepVisual* minimizes the coding effort and facilitates a better understanding of the correspondence of the DL system and code intuitively.

B. Layout of DeepVisual

The layout of *DeepVisual* is illustrated in Fig. 5. *DeepVisual* is consisted of four main panes: *Datasets*, *Neural Network Builder*, *Compile&Fit* and *Code*. The first three panes (Fig. 5(a)-(c)) visualize the different stages in the programming process, and the last pane (Fig. 5(d)) displays the generated code.

Datasets pane. *DeepVisual* provides with 7 popular DL datasets. After a developer chooses a dataset, the code of importing the targeted dataset along with the preprocessing method is automatically generated and displayed in code pane. The chosen dataset also determines some parameters in following neural network construction procedure. For example, once the user chooses MNIST (*i.e.*, a dataset for handwritten digits recognition), the *input shape* of the neural network is automatically set as $(28, 28, 1)$, and the *classification category size* is set to 10.

Neural Network Builder pane. The layout of the Neuron Network Builder is illustrated in Fig. 5(b). Neuron Network Builder pane is utilized to build a neural network via GUI programming. *DeepVisual* represents each kind of layers as a component (*i.e.*, convolutional layer, max pooling layer, etc.). These components are listed in the layer menu. A developer can drag, drop, and connect them in draw pane to design and construct a neural network. The attribute display window on the bottom right of *DeepVisual* shows the corresponding attribute values of the clicked component, *e.g.*, neuron number, activation function and so on.

Compile&Fit pane. In Compile&Fit pane, which is shown in Fig. 5(c), a developer configures various parameters for compiling the DL model and fitting the model to the dataset, such as *optimizer*, *loss function*, *epoch*,and *batch size*.

Code pane. *DeepVisual* generates the code of a DL system and displays it in Code pane. Any slight modification made in the other 3 panes will in terms dynamically affect the generated code. Besides, a developer can either export the code as a Python file or import a training code file in targeted DL framework to visualize its architecture design and training configurations in Neural Network Builder pane.

C. Visual Programming in DeepVisual

The current state-of-the-art DL frameworks often follow the modular design, where each kind of layer is often defined as a separate module. To construct a deep neural network by code, a developer invokes the corresponding module of a layer, which is often consisted of 3 parts: instance creation, attribute values of the layer and the name of the connected layer.

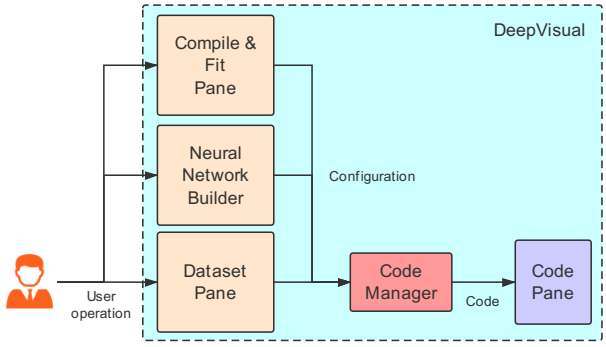


Fig. 3. The workflow of generating code from graph structure in DeepVisual.

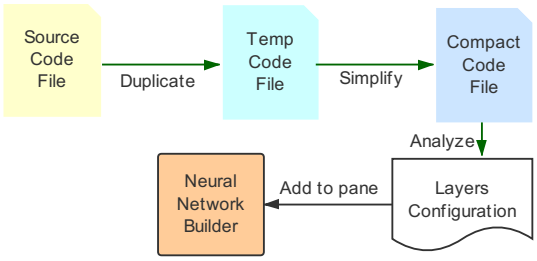


Fig. 4. The workflow of generating graph structure from code.

To provide visual development support, we use different components to represent each kind of layers' instance, and each component is associated with a piece of the corresponding calling code. Fig. 2 illustrates an example of graphic representation in Keras. Each component has a node in each side of its rectangular shape to connect each other. A component can be connected only once under the definition of layer connection in the neural network. Then, we fulfill the essential features in *DeepVisual*: generate code from graph structure and generate graph structure from code.

1) Generate code from graph structure: Fig. 3 highlights the workflow of generating code from a graph structure. In *CodeManager*, a linked list is used to manage the data, each node of which has the structure of $[Component, Code]$ or $[Code]$ (*e.g.*, data preprocessing code). When a developer drags-and-drops a component from layer menu into in Neural Network Builder pane, a new component is created and assigned with an index. Then, the component is sent to *CodeManager*, in which its attribute is analyzed and the corresponding code is called and added into the list.

To establish the data flow connection of two components, a connecting edge is created. Then the edge and the corresponding starting and ending components are both sent to *CodeManager*, in which the linked list aftermentioned is traversed to locate the node of the ending component. The parameter *(Name of Starting Component)* is further added at the end of its layer definition code. After locating the node of the starting component, it is inserted at the front of the node of the ending component. Besides, The nodes $[Code]$ with importing packages or data pre-processing code are at the front part of the list, while the nodes with Compile&Fit and evaluation code locate at the back part.

When a user edits the attribute display window on the bottom right of *DeepVisual*, the chosen component and its edited attribute array are sent to Code Manager. After locating the corresponding node, the attribute array in the code is updated.

At last, Code Manager generates the code according to the linked list sequence, and the generated code is displayed in Code pane.

2) Generate graph structure from code: The workflow of graph structure generation by code is illustrated in Fig. 4. In the Code pane, a developer imports DL source code from storage. To avoid accidentally corrupt a source file, *DeepVisual* duplicates the code file, performs code simplification by removing the code fragment that does not contain the layer information, and analyzes the simplified code to extract the layers configuration, which often includes all the information about the DL system. *DeepVisual* then creates components managed as a graph data structure and generates the graphical structure of the neural network. After that, the visualized network structure with configurations is displayed in the Neural Network Builder.

D. Implementation

As a proof-of-concept, we implemented *DeepVisual* as a PyCharm plugin based on Keras 2.2.2. The current version of *DeepVisual* supports almost all of the Keras layers: 9 main categories and 55 different kinds of layers in total. We select PyCharm as the implementation platform because it is one of the most widely used Python IDE at present. Our current implementation selects Keras as the target DL platform based on several considerations. First, Keras is among the most popular deep learning framework [7] providing high-level APIs on top of several other DL frameworks (*e.g.*, TensorFlow, CNTK, and Theano [8]). The highly compact design and layer component correspondence make it more accessible for DL beginners, which, we believe, could be a suitable starting point.

Even though our proposed visual programming technique is general, and *DeepVisual* could be easily extended to other DL frameworks. To facilitate further research on DL visual programming, we make *DeepVisual* open source and publically available, which could be accessed at https://github.com/SaikaQH/DeepVisual.

III. Use Cases

In this section, we demonstrate the usefulness of *DeepVisual* on two typical use cases (see Fig. 5). The first demonstration shows how to leverage *DeepVisual* to perform visual development of DL software for a typical task (*i.e.*, image handwritten digit recognition). We intend to construct a LeNet-5 DL model and train it on MNIST. In the second example, we use *DeepVisual* to extract and generate the neural network

structure of a DL software from a Keras source code file.[1] We made a video to demonstrate the use cases, which can be accessed at https://youtu.be/KQhylv38atk.

Image Recognition DL Software Development. In this example, we use *DeepVisual* to visually program a LeNet-5 DL model and train it on MNIST. First, we launch *DeepVisual* in PyCharm by clicking DeepVisual button (in PyCharm-Tools menu). *DeepVisual* displays Dataset pane at the beginning and we choose MNIST dataset as shown in Fig. 5(a). Then, we turn to Neural Network Builder pane and, according to the structure of LeNet-5, we drag-and-drop *Covolutional*, *Maxpooling*, *Dense* and *Flatten* components from the layer list displayed at the top right of the window (see Fig. 5(b)). We click a component's node in the side of its rectangular shape to create a connecting line to connect another component. Then, we click each component and input its corresponding attribute values at the bottom right of the attribute window. Finally, as shown in Fig. 5(c), we set the compiling and fitting parameters in Compile&Fit pane to finish the visual programming process.

Based on the obtained DL software design, the training program is automatically generated and displayed in Code pane (see Fig. 5(d)). We can either copy-and-paste the code to our Python file or export the code as a Python file (via Code pane-File) directly.

Neural network structure reconstruction. In this example, we first download the MLP for multi-class classification training code from Keras homepage. After launching *DeepVisual* in PyCharm, we switch to Code pane and import the code file. After importing, the corresponding neural network structure is automatically generated in Neural Network Builder pane (see Fig. 6(b)). We click each component to check its configuration information at the bottom right of the attribute window.

During our hands-on evaluation, we observe that *DeepVisual* might have great potentialities and exhibit several benefits for DL system development:

- A user can design and implement the DNN at the same time, and the corresponding code is generated automatically.
- Visualizing the whole programming process during the DL system development can greatly reduce manual efforts.
- Development in *DeepVisual* is often efficient and avoids many programming typos as those in writing source code.
- The graphic structural representation of the neural network facilitates a user to understand the bidirectional correspondence between neural network and source code.
- Automated neural network structure reconstruction and visualization by importing source code can potentially reduce the barriers of learning DL for beginners and assist them to get started quickly.

IV. Related Work

Besides TensorFlow playground, TensorBoard and other commercial visualization tools mentioned in Section I, DeepGraph [9] is the most relevant work to ours. In particular, DeepGraph is designed to facilitate understanding DL models through visualization. It works after the training process and generates the data flow graph of the targeted DL network to observe how a neural network works. DeepGraph also supports code mapping to help a user to better understand the source code of TensorFlow. Different from DeepGraph, our *DeepVisual* focuses more on GUI programming to develop DL systems and uses the structure graph to describe DNN rather than a data flow graph in DeepGraph. The features of DeepGraph complement *DeepVisual* and the integration of DeepGraph into *DeepVisual* will be our future work.

[1]The source code file we utilize in this example is a neural network for multi-class classification from Keras homepage at https://keras.io/getting-started/sequential-model-guide/.

V. Conclusion and Future work

In this paper, we presented *DeepVisual*, a visual programming tool for deep learning software development. *DeepVisual* allows a developer to design and implement the DL software visually. Furthermore, *DeepVisual* can also reconstruct the neural network structure of the DL software by importing the DL source code.

In the future, we plan to add more features such as components folding and zooming in Neural Network Builder pane. We also plan to provide support for other mainstream DL frameworks like Tensorflow and PyTorch. Furthermore, the current version of *DeepVisual* does not support automatic preprocessing for the user-customized dataset, which is another essential feature to enhance in our future work.

Acknowledgement

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

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[2] F. Chollet *et al.*, "Keras: The python deep learning library," *Astrophysics Source Code Library*, 2018.

[3] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, 2017.

[4] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 2135–2135.

[5] D. Smilkov, S. Carter, D. Sculley, F. B. Viégas, and M. Wattenberg, "Direct-manipulation visualization of deep networks," *arXiv preprint arXiv:1708.03788*, 2017.

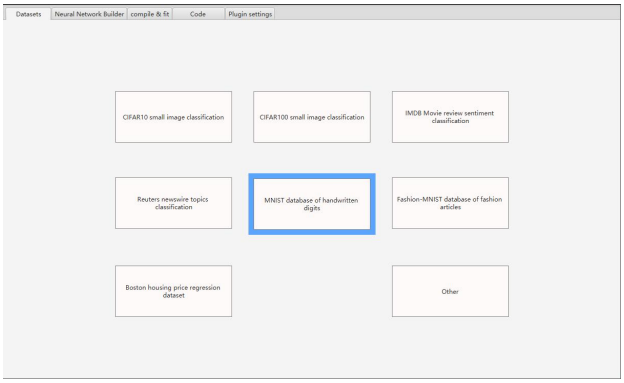
[6] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 1–12, 2018.

[7] J. Hale, "Deep learning framework power scores 2018," https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a/, accessed September 19, 2018.

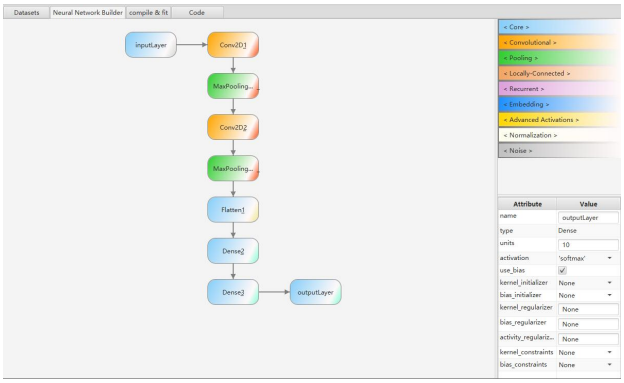
[8] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th Python in Science Conf*, vol. 1, 2010.

[9] Q. Hu, L. Ma, and J. Zhao, "Deepgraph: A pycharm tool for visualizing and understanding deep learning models," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018.

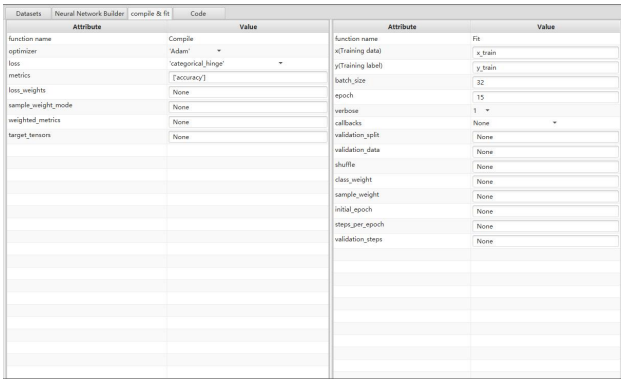
APPENDIX



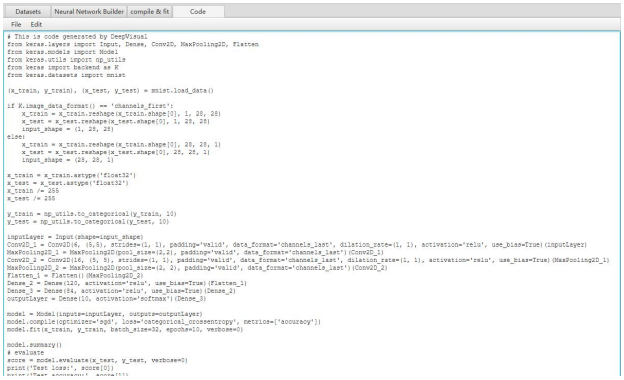
(a) Select MNIST dataset in Datasets pane



(b) Construct LeNet-5 in Neural Network Builder pane.

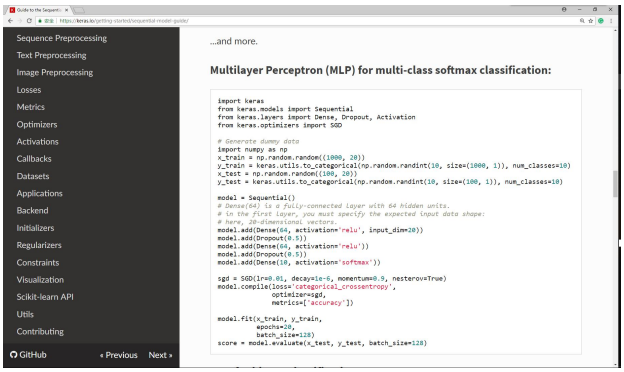


(c) Configure hyper-parameters in Compile&Fit pane.

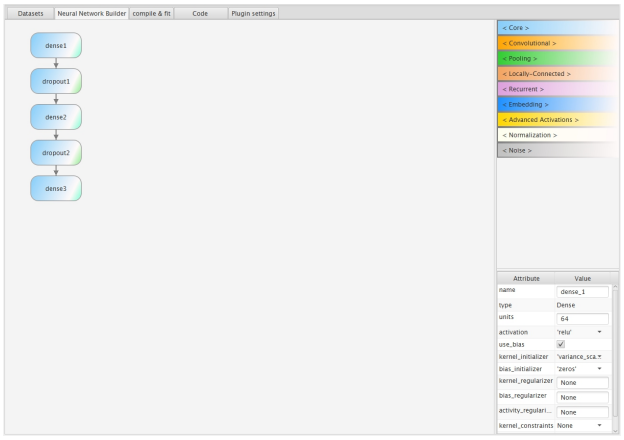


(d) Display the generated code in Code pane

Fig. 5. Develop the image recognition DL software with DeepVisual.



(a) Download DL code from Keras homepage.



(b) Display the generated neural network structure in Neural Network Builder.

Fig. 6. Generate deep neural network structure by importing Keras code.