Live Programming Environment for Deep Learning with Instant and Editable Neural Network Visualization

Chunqi Zhao
The University of Tokyo, Japan
shunnki.chou@is.s.u-tokyo.ac.jp

Tsukasa Fukusato
The University of Tokyo, Japan
tsukasafukusato@is.s.u-tokyo.ac.jp

Jun Kato
National Institute of Advanced Industrial Science and Technology, Japan
jun.kato@aist.go.jp

Takeo Igarashi
The University of Tokyo, Japan
takeo@acm.org

Abstract

Artificial intelligence (AI) such as deep learning has achieved significant success in a variety of application domains. Several visualization techniques have been proposed for understanding the overall behavior of the neural network defined by deep learning code. However, they show visualization only after the code or network definition is written and it remains complicated and unfriendly for newbies to build deep neural network models on a code editor. In this paper, to help user better understand the behavior of networks, we augment a code editor with instant and editable visualization of network model, inspired by live programming which provides continuous feedback to the programmer.

2012 ACM Subject Classification Information interfaces and presentation (e.g., HCI) → User interfaces; Software Engineering → Programming Environments

Keywords and phrases Neural network visualization, Live programming, Deep learning

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Acknowledgements This work was supported by JST CREST JPMJCR17A1.

1 Introduction

In recent years, deep neural network (DNN) model has garnered tremendous success in various application domains such as Image Processing (IP) and Natural Language Processing (NLP) research. From well-structured models, effective features can automatically be extracted without selecting manual-designed filters. Thus, deep learning has become a competitive solution for most traditional application domains, as well as some new areas where it shows the possibility such as computer graphics and robotics researches.

Deep learning programming, however, distinguish itself from conventional programming with some unique features. That is, deep learning algorithm is not to provide solutions to any specific applications, but to provide a way to extract features from data then optimize the parameters in a huge matrix with the features and the pre-defined constraints, and finally makes the optimized matrix a solution towards the application. Thus, rather than determining solution details for specific problem by programmer in conventional programming,

deep learning programming is to design a general way to search for solutions in various training data.

Many DNN framework libraries [?, ?, ?, ?] provide full-stack deep learning toolbox from preliminary layer definition, GPU-accelerated training optimizer to evaluation function. And nearly all the DNN frameworks follow the same coding style: (i) users use a separated file or block of code to define the network, then (ii) imports the network as an instance in the training control module. The problem is, the training process is resource-consuming. Before the actual training, the user needs to design the DNN architecture, load and pre-process dataset, give the hyper-parameters and determine how data goes through the system. If the shape of data in computing doesn't match the defined layer's parameters, the training process will fail to start. However, in current practice, if the user don't want to debug by running the training script, the only way to check the computed data is in a desired shape or not, is to validate data shapes on scratch paper line by line. Hence instant visualization during programming will be a more timely feedback for the programmer. Some DNN frameworks provide visualization of network structure as a node-link diagram. But they show such visualization only after the code or network definition is written, or even after the training phase is completed. Such visualization is not very helpful for newbies to build deep networks on a code editor.

Therefore, we present a novel web-based live programming environment specialized for neural network programming by integrating an instant and editable visualization into a standard code editor. The system is aware of which part of codes belongs to the layer definition and which part belongs to the data flow control. For the layer definition codes, when the user types in one line that contains the configuration of a neural network layer, the system understands the context and instantly create the layer's visualization at the visual panel; And for the data flow control part, we provide a bidirectional mapping between codes and visualization: we visualize the defined layers as candidates, and enable the user to build codes by dragging dropping their visualization, or vice versa. This paper reports our design principles and some implementation details, as well as work in progress.

2 Related Work

Neural Network Visualization. Chainer [?] and Keras [?] has built-in visualization functions. However their display are too difficult to grasp the true important information of a neural net. Wongsuphasawat et al. [?] proposed a plugin in TensorBoard to visualize neural nets with data flow direction, but also particularly problematic that programmers may be confused in choosing the best view to show the important information. While several open source tools [?, ?, ?, ?] to visualize the networks are designed, they are focus on the stage after the user finish editing their source code, or even after the model's training. We therefore visualize networks at coding time, and propose the first tool to enable user programming deep learning code with living visual feedback.

Live Programming Environment. Live programming is a technique to provide the programmers with continuous feedback about the current program for understanding the behavior of the under-development program. One approach is to use explicit representations such as images, sounds or video [?, ?, ?] instead of textual information such as stack traces. The other approach is to directly display the value of program variables. For example, Python Tutor [?] divides the execution process into several steps of the program, and continues executing the under-development program. Kanon [?] enables to synchronously analyze data structure (e.g., linked list) without actually executing the program and update the data

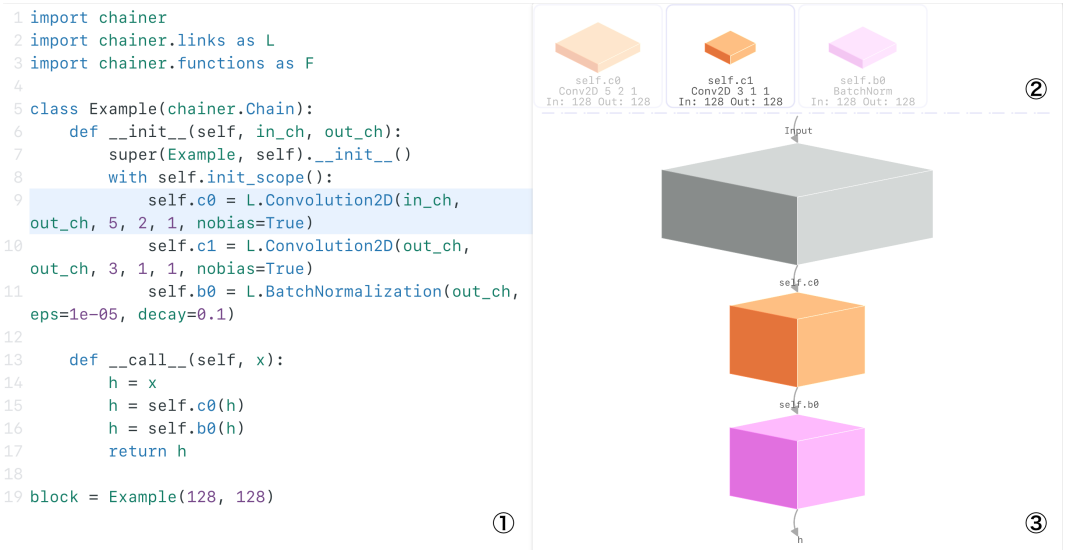

Figure 1 The screenshot of our system's first prototype. The interface is made up of: (1) code editor, (2) grid-like candidate region where layers defined in ___init___ are listed here, (3) data cubes are visualized in the form of a directional graph, whose direction is given in ___call___.

nodes graph. Our work builds on the live programming approach to develop deep learning programs while considering the characteristics of the current program.

3 Design Overview

3.1 Interface

Our interface (See Figure 1) consists of (1) the standard code editor to write DNN programs (the left-hand side) and (2) the visualization panel to display layer/dataflow (the right-hand side).

3.2 Design Principles

Based on the background of deep learning visualization tools, we design the system with the principles below.

First, and also the key idea to support our design is, **the visualization should be presented during coding**. Our motivation is to help user understand the network structure and simplify the debugging of deep learning programming. Naturally, the "live" programming manner that merges "coding" and "test" into a single phase makes the program developing a shorter loop.

Secondly, **besides the code editor, a graphical editor can be more helpful to a newbie**. Our target user is not a deep learning expert - that means, he/she might be a newbie in programming, or a programmer without deep learning programming experience. The utilization of graphical editor instruct the user, especially the newbie user, to build a neural network in a more intuitive way, just like drawing the network structure on scratch paper.

Finally, **to visualize the network in a most common, but interactive way**. Referring to the visualizations of deep neural networks from published papers, we think box-like

visualization represents data's shape best. Layers and data are the two main factors in neural networks visualization, but if treated the same, the view will seem in a clutter. We believe that the suitable practice to visualize the network, is to have the two factors separated somehow.

3.3 Assumptions

To better locate our system's feature and the usage scenarios, we make assumptions as below:

- The DNN program is written in Chainer framework. In Chainer, the standard style to define a neural network is the code snippets in Figure 1. The neural network is defined using a Class that extends from chainer.Chain. The class, consists of two important functions: ___init___ gives all the layers' type and shape in the net, while ___call___ determines the order of layers.
- The user will define ___init___ first, then use the defined layers to further give the order of layers that the input data will go through in ___call___.
- The user will create a instance of the DNN he/she defined at the end of the program. This is for the convenience of syntax check.

4 Implementation

The whole system can be decomposed into frontend and backend. The frontend collects user's inputs in code and graphical editors, and the backend acts as a code parser.

4.1 Abstract Syntax Tree Parser in Backend

We set up a Python parser to process the code submitted from browser. The backend program checks the syntax of code, then parses it into abstract syntax tree. Since what we are interested in is the defined layers and data flow information, the program will distil the ast into a much smaller object that only contains the necessary information for user. Finally the object is returned to the browser for further processes.

4.2 Collaborated Code Editor and Visual Panel in Frontend

The proposed system can bidirectionally update the code editor and the visual panel, so this section describes the behaviors in each direction separately.

1) Code ⇒ Visualization. After the browser receives the extracted net information from the backend, it snapshots this frame and visualizes it. In the case where only ___init___ is given, layers will be visualized in (2) of our interface as cube models whose length and width represents the kernel size. Since connections are not defined, all cubes are individually packaged in a cell and listed in a grid-like structure. If ___call___ is also given by the users, layers picked in ___call___ will become transparent and its name will show on the lines between data cubes in (3) of the interface.

2) Visualization ⇒ Code (In progress). We consider that to define a new layer, coding is a much more efficient means rather than graphical editing, thus we disable the synthesis of layer definition from visualization operation. The more encouraged way to edit codes from the visual panel in our system is, with the defined layers cubes and the input data node, the user drag the layer, and drop at the data he/she want to process using this layer, in this way, the new line of codes in ___call___ as well as the last data in the dataflow will be

synthesized one by one. For deletion, the mapping between code and visualization will still be bidirectional.

5 Limitation and Future Work

The current system is still a prototype, so we only support Chainer and the most common layers at this stage. Support for more frameworks and layers will make our system fit more situations. Besides, we consider adding an argument hinter to simplify layer argument configuration. There also remains space to make the code editor and the visual panel collaborate better, like more simultaneous visual feedback. In addition, evaluation such as user study will be performed to validate the effectiveness of the proposed system.

References

1 Chainer. `https://chainer.org/`. Accessed: 2019-06-05.
2 Hiddenlayer. `https://github.com/waleedka/hiddenlayer/`. Accessed: 2019-06-05.
3 Keras. `http://keras.io/`. Accessed: 2019-06-05.
4 Nn-svg: Lenet- and alexnet-style diagrams. `http://alexlenail.me/NN-SVG/LeNet.html`. Accessed: 2019-06-05.
5 Plotneuralnet. `https://github.com/HarisIqbal88/PlotNeuralNet`. Accessed: 2019-06-05.
6 Pytorch. `https://pytorch.org/`. Accessed: 2019-06-05.
7 Tensorflow. `https://www.tensorflow.org/`. Accessed: 2019-06-05.
8 Tensorspace.js. `https://tensorspace.org/`. Accessed: 2019-06-05.
9 Philip J Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceedings of the 2013 ACM SIGCSE*, pages 579–584. ACM, 2013.
10 Jun Kato. Visionsketch: Gesture-based language for end-user computer vision programming. In *Proceedings of the 2013 ACM SIGPLAN*, 2013.
11 Jun Kato, Sean McDirmid, and Xiang Cao. Dejavu: Integrated support for developing interactive camera-based programs. In *Proceedings of the 2012 ACM UIST*, pages 189–196. ACM, 2012.
12 Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. Eyepatch: prototyping camera-based interaction through examples. In *Proceedings of the 2007 ACM UIST*, pages 33–42. ACM, 2007.
13 Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. Live, synchronized, and mental map preserving visualization for data structure programming. In *Proceedings of the 2018 ACM SIGPLAN*, pages 72–87. ACM, 2018.
14 Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):1–12, 2017.