ODEN: Live Programming for Neural Network Architecture Editing

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

I-Chao Shen
jdilyshen@gmail.com
The University of Tokyo
Tokyo, Tokyo, Japan

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

Jun Kato
jun.kato@aist.go.jp
Japan National Institute of Advanced Industrial Science and Technology (AIST)
Tsukuba, Ibaraki, Japan

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

ABSTRACT

In deep learning application development, programmers tend to try different architectures and hyper-parameters until satisfied with the model performance. Nevertheless, program crashes due to tensor shape mismatch prohibit programmers, especially novice programmers, from smoothly going back and forth between neural network (NN) architecture editing and experimentation. We propose to leverage live programming techniques in NN architecture editing with an always-on visualization. When the user edits the program, the visualization can synchronously display tensor states and provide a warning message by continuously executing the program to prevent program crashes during experimentation. We implement the live visualization and integrate it into an IDE called ODEN that seamlessly supports the "edit→experiment→edit→⋯" repetition. With ODEN, the user can construct the neural network with the live visualization and transits into experimentation to instantly train and test the NN architecture. An exploratory user study is conducted to evaluate the usability, the limitations, and the potential of live visualization in ODEN.

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**.

KEYWORDS

deep learning programming, neural networks, programming experience

ACM Reference Format:
Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces (IUI '22), March 22–25, 2022, Helsinki, Finland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490099.3511120

1 INTRODUCTION

Deep learning (DL) has attracted considerable attention in many research areas, such as computer vision and natural language processing. Many programmers are enthusiastically testing deep learning algorithms on their own dataset to obtain new insights, who do not necessarily have the mathematical and theoretical background required to understand how machine learning works [7]. In practice, programmers rarely build DL models from scratch and instead utilize libraries such as Tensorflow [1, 2] and PyTorch [20]. These libraries provide high-level APIs to build a DL model while retaining the flexibility to customize the DL model in detail. The learning resources for novice DL programmers (e.g., blog posts and GitHub repositories) also adopt these libraries in their code examples. Following these resources, novice DL programmers apply these libraries in their model development and repeatedly transit between *neural network (NN) architecture editing* and *experimentation* [21] phases to improve the performance of their DL models by trial and error.

The transitive development manner results from the experimental nature of the ML application [7]. However, recent studies show that novice DL programmers often struggle in DL environment setup and model implementation issues that abort the transition [3, 36]. Among these issues, tensor shape mismatch is a typical reason that causes program crashes [36]. In addition, there are specific needs for network architecture editing in DL model development, as shown in Yan's formative study [33]. For instance, one of their participants mentions that *"based on our data, we may change our (network) structure and add few more layers behind or in front of the original network"*. This process potentially introduces tensor shape mismatch errors in novice programmers' DL programming, which prohibit novice users from the smooth transition between the *NN architecture editing* and *experimentation* and make the development cumbersome (the details will be described in section 3).

This paper presents a live NN architecture editor design to help solve shape mismatch errors in network architecture editing. We introduce live programming techniques into NN architecture editing

IUI '22, March 22–25, 2022, Helsinki, Finland

ACM ISBN 978-1-4503-9144-3/22/03…$15.00
https://doi.org/10.1145/3490099.3511120

and enable an always-on tensor-semantic visualization alongside a traditional text-based code editor. We adopt Projection Boxes [17] in our live programming environment design and adjust the visualization for synchronously displaying tensor shape information semantics. We provide three view modes for showing tensors in different dimensions. The editor also supports NN architectures in a nested form which is common in practice. We implement the live NN architecture editor in an integrated development environment called *ODEN* to show its value in the actual *NN architecture editing* and *experimentation* transition. Unlike other IDEs designed to assist general-purpose programming [31], ODEN specifically helps the initial and exploratory DL model development. The programmer first edits the NN architecture in the live NN architecture editor, then instantly trains and tests the edited network architecture using an integrated and sandboxed experimentation panel. We ran an exploratory user study that involved seven participants in evaluating the usability, the limitations, and the potential of the live NN architecture editor. During the user study, the participants were requested to improve the model performance based on an existing NN architecture and a dataset. We observed their developing strategies and noticed several actual usage scenarios with ODEN. The participants, who were all identified as novice DL programmers, thought the live NN architecture editor helps diagnose and solve tensor shape mismatch while editing their DL programs.

In summary, we highlight the contributions of this work as below:

(1) A live programming environment for solving tensor shape mismatch problems. It synchronously analyzes and visualizes in-edit neural network programs. The live visualization can inspect not only linear networks but also nested networks with multiple sub-networks.
(2) An IDE implementation called ODEN that integrates the live NN architecture editor and a sandboxed experimentation panel to support the seamless transition between the *NN architecture editing* and *experimentation*.
(3) An exploratory user study to assess the usability, the limitations, and the potential of the live NN architecture editor and the usage scenarios found during the user study. I

2 RELATED WORK

2.1 Hurdles to access Deep Learning

Recent studies have investigated obstacles that non-expert DL users face, and their findings inspire us to address these obstacles. Cai and Guo surveyed 645 software engineers for their desire to learn machine learning and the hurdles they face during their learning [7]. They found that novice users desire paired conceptual tutorials and code examples to help improve their understanding of complex and esoteric ML theories. One response in their survey is, "*It was nice [...] to have the small code demos that you can edit and run right there. It really helps basic understanding.*" Additionally, they also found that the convenience of modifying an existing model and the experimental nature in developing an ML application should be emphasized in a tool to support novices. These findings motivated us to make a scaffolding tool that is easy enough to run, modify from a template, and experiment in different model-building processes.

Zhang et al. [36] analyzed and categorized the DL-related questions from StackOverflow and indicated that frequent implementation failures of DL applications and program crashes stop novice users from quick experimentation. Among the program crashes, tensor shape mismatch is a typical error type. Compared to other typical program crash reasons, the tensor shape mismatch error is more complicated and cannot be addressed with static code checks. We will discuss this in the next section. Patel et al. identified three significant obstacles to applying ML as a tool in software development, including the difficulty of using ML in an iterative and exploratory way [22]. Yang et al. found that most non-experts with programming skills prefer interactive ML tools that take the form of a code editor compared to expert users because "*they can more easily re-use online scripts and ML solutions*" [34].

We designed our tool as an extended code editor to address these needs: the tool should help reduce the program crashes in the DL application development; it should support importing existing code templates into its interface, then allow the user to modify and check the model program interactively; it should consist of mechanisms to shorten the test and debugging cycle of DL programs; the usage scenario of it should consider the exploratory and experimental nature of DL development.

2.2 Deep Neural Network Visualization

Many visualization tools have been proposed to support different subprocesses during the development of a DL application. Tensorboard [32] is a popular tool to help visualize a running DNN instance after the in-editor stage and the metrics such as accuracy and training loss. Kanit et al. also conducted a study among DL programmers, and found that programmers commonly draw tensor shape diagrams before implementing. This phenomenon inspires us to bring in-edit visualization into DL programming. Neural Network Console [27] works as a visual programming tool for the entire process of DL application development, including NN architecture building, training monitoring, and testing. Some other tools aim at helping interpret a pre-trained model or the general behavior of a DNN, such as LSTMVis [28] and TensorFlow Playground [26]. Our system differs from these tools because we support on-the-fly visualization during text-based code editing.

To our knowledge, Skyline [35] is the most closely related to our work. As an in-editor DNN profiling and visualization tool, Skyline provides computational performance information such as memory consumption and throughput during the code editing phase. It leverages an always-on runtime analyzer to re-profile the user's code every time the user alternates the current program. We also adopt such a live programming environment. Nevertheless, our system differs from Skyline in three perspectives. First, while Skyline serves experienced DL programmers, our system mainly targets beginner DL users. Second, Skyline only assists users in the NN architecture design process, while our system further allows users to go forward to experimentation and back to editing. Finally, Skyline implements computational performance profiling in an editor to help solve computational performance issues. On the contrast, our system serves as a scaffold that prevents tensor shape mismatch errors and enables novice users to experiment with the edited NN architecture instantly.

2.3 Live Programming Environment

The research on live programming has a long history, and many properties and applications have been proposed. Hancock [9] introduces several essential properties of live programming; different levels of programming liveness have been categorized [29]; many live programming environments have been implemented in general-purpose programming [8], data structure programming [18], and micro-controller device development [14].

Projection Boxes [17] introduces non-intrusive on-the-fly reconfigurable visualization into general-purpose live programming. It leverages full-semantic, line-related, and auto-shrink projection box visualization alongside the traditional code editor to provide program visualization meanwhile avoiding information overload. We built our live visualization inspired by Projection Boxes (i.e., mode switching, features to prevent information overload, line-level program semantics) but chose how and what to display that suits the DL programming process.

3 BACKGROUND

In this section, we briefly introduce the background of DL programming practice.

3.1 The Practice of DL Application Development Using Libraries

To define a DNN, a programmer needs to write a program that assembles a series of mathematical functions into a sequence. The sequence of mathematical functions is called a *network*, and each mathematical function is known as a *layer* in the network, which may maintain several internal parameters, i.e., *weights*.

Once the programmer has assembled the network using several layers, the programmer writes scripts to define the *training* process, where the sequence learns prediction from the given data. The training program instantiates the DNN into a *model*, then, during training, iteratively updates the model weights learning from *batches* of input data. In each *iteration* of the training process, the model predicts from the input data batch and computes the error by comparing the model prediction with the ground truth. The program then optimizes the model weights gradually to decrease the prediction error. In this process, layers receive and emit data in the form of a multi-dimensional array, e.g., for an image as the input into the network, the array consists four dimensions: `batch size`, `channel size`, `height`, and `width`. The multi-dimensional array is called *activation* or *tensor* in the context of DL programming.

After the training process ends, the programmer *evaluates* the trained model performance and iteratively improves it by mutating hyper-parameters, training data, and network architecture.

3.2 Define-by-Run and Structured Network Code

Most modern DL libraries such as Chainer [30], PyTorch [20], and Tensorflow Eager [2] adopt the idea of define-by-run in their API design. Define-by-run means that the program instantiates layers as a property of a `Module` or an `object`, then directly calls the instance to process tensor. Oppositely, the classic define-and-run idea requires the programmer to prepare a network definition scheme outside the program, limiting network editing flexibility. If we pick PyTorch as the tool to program the DNN in a define-by-run manner, a typical pattern to structurally program a network is defining a `Module` class and assigning layers as its properties. The snippet[1] may look like:

Listing 1: PyTorch define-by-run module snippet example

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.c = torch.nn.Conv2d(3, 48, kernel_size=11,
            ↪ stride=4, padding=2)
        self.b = torch.nn.BatchNorm2d(48)
        self.a = torch.nn.ReLU()
    def forward(self, x):
        h = self.c(x)
        h = self.b(h)
        h = self.a(h)
        return h
n = Net()
x = torch.randn(1, 3, 128, 128)
y = n(x)
```

Note that the whole network and each independent layer both inherit the class `Module`, which means that the network can also be regarded as a layer. This characteristic makes it possible to reuse small networks to construct a more extensive network with a nested architecture.

3.3 Common Runtime Errors in DL Programming

Similar to general-purpose programming, bugs in DL programming can also be identified as explicit or implicit bugs. Implicit bugs will not produce errors during program execution, but cause symptoms like, e.g., abnormal training or low prediction accuracy. Explicit bugs crash the program and abort the training or evaluation process.

According to Zhang's research on DL-related questions collected from StackOverflow [36], the explicit bugs in DL programming can be categorized into three factors: shape mismatch (inconsistency), numerical error, and CPU/GPU incompatibility. Such bugs are most frequently reported in DL application-related questions but hardly appear in conventional non-DL applications.

In this paper, we address the explicit bugs, a.k.a. runtime errors, and focus on shape mismatch errors because the shape mismatch errors often result from the repeated NN architecture mutation, which is typical in the iteration of DL model development. Moreover, the shape mismatch error has its particular characteristics, making it unable to be solved with static code check. The shape mismatch error is explained as follows.

Layers in a network are defined with several arguments and can only receive tensors in a specific shape. If the arguments of

[1] At the last line of Listing 1, `n(x)` forwards the argument `x` to the other hook functions and the defined `forward` function. For simplicity, readers can think `n(x)` is equivalent to `n.forward(x)`

a layer mismatch with the input tensor shape, the execution will be aborted and the program interpreter throws an error. E.g., in the image classification task, when a convolution layer receives an input tensor and produces a tensor with noninteger `height` or `width`, a tensor shape mismatch error will happen. Note that the tensor shape must be integers.

The acceptable input tensor shape of a layer depends on the layer type. If we take PyTorch library as an example, `nn.Linear(120, 84)` means the linear layer can only receive a 1D tensor of length 120 and transform it into a 1D tensor of length 84. In this case, the acceptable input tensor shape is unique. Layers such as convolution and pooling layers will accept a specific pattern of shape rather than a uniquely defined shape in most cases. To summarize, in many layers, the input and output shapes are not solely determined by the layer definition itself but are also influenced by the real-time input tensor shape. In this sense, checking tensor shape mismatch becomes cumbersome because the programmer needs to trace the tensor shapes from the start to the end of a neural network. These characteristics make the programmer have to judge whether a tensor is acceptable to the current layer and maintain the mental image during the NN architecture editing. Maintaining the mental model towards the NN architecture will become extremely difficult as the network architecture scale becomes larger, especially for novice programmers. Additionally, these characteristics make the static code checking approach difficult to cover all layer types.

4 LIVE NEURAL NETWORK ARCHITECTURE EDITING

Starting from the motivation of preventing program crashes resulting from shape mismatch errors in NN architecture editing, we designed an always-on live visualization of tensor shapes to enhance a traditional text-based code editor. Our design originates from the NN architecture editing background in section 3 and our own experience. This section summarizes the objectives that the live visualization should reach to serve the background of DL programming introduced in section 3. For each objective, we describe how we correspondingly design the functions.

Tensor-related semantics for NN architecture editing. When programmers face shape mismatch errors, they desire to view each layer's input and output tensor shape in the neural network. The first objective of the live NN architecture editing is to make the tensor shape transformation transparent to programmers in the editing stage rather than the program execution stage. Therefore, the live visualization starts by tracking a NN's tensor-related semantics. In contrast with a general-purpose program, a PyTorch program computes new tensors in the `forward` method of the network module. We use the state collecting semantics in this specific method: at each line of the `forward`, we trace all the input and output tensor shapes related to the line. We visualize the layer type and input output tensor shape while ignoring other information such as the parameters of layers.

Code-connected box visualization that informs tensor shape transformation. The live visualization should maintain the connection with the relevant code so that the user can quickly locate the tensor shape transformation in the code editor. Figure 1 shows the tensor state box visualization. Each box represents one line in the program, which floats at the right half of the editor and is connected to the right end of the corresponding code by a straight line. A box shows the input tensor shapes, the layer type, and the output tensor shapes in three rows, from top to down (see Figure 2). Tensor shapes are shown with visualization on the left and the shape information on the right. Layers are represented using an arrow together with the type of the layer. Besides, an extra red dot will show at the left top of the box if any error occurs in the lines before the corresponding code of the box to indicate that the information in the box has been outdated (see Figure 2(b)). To avoid the common information overload issue in live programming, we adopt three visualization techniques from [17]: the auto-shrink effect, the fisheye effect, and the transparency effect.

Configurable tensor shape visualization. Layers may require different types of tensor visualization. For example, in terms of the input/output tensors of a convolution layer, the user needs the tensor shape information of all dimensions, so the tensor visualizations are best represented in 3D. In contrast, a linear layer's input/output tensors are best described in 1D because linear layers only receive flat (1D) tensors. To address this desire, we enable the programmer to change the box view modes using a dropdown list at the top bar of the interface or a keyboard shortcut. There are three modes in the live visualization: a) Cuboid mode that draws tensors using 3D-boxes, and the shape is displayed as a text in the form `(channels, height, width)`. Cuboid mode is the default mode, and it shows all information about the tensor shape; b) Rect mode that draws tensors using 2D-boxes, and the shape is in the form `(height, width)`. This mode does not show the channel size of tensors directly but encodes it as the color of 2D-box; c) Line mode that draws tensors using 1D straight lines, and the text of the shape information is the total element number of the tensor. The live visualization will automatically choose the default view mode for different tensors. If the tensor is a 3D tensor, the live visualization selects the cuboid mode by default; if the tensor is 2D, then the rect mode; and if the tensor is 1D, then the line mode.

Nested network architecture inspection. In the early stage of our design process, we conducted a casual interview involving four DL experts. We showed the basic features of our live NN architecture editor and listened to their needs as expert users. From their perspective, experienced DL programmers commonly need to define some child networks consisting of repetitive layer sequences (e.g., *ConvBatchNormReLU* blocks) then reuse them in a larger parent network. Therefore, there is another objective that the live visualization should support the programmers inspecting those nested NN architectures. Just like the code navigation function of IDEs for general-purpose programming that hierarchically shows functions and classes, the live visualization introduces a similar multi-level inspector to help check the child network details inside a parent network. As shown in Figure 3, the example program defines a parent network `Net` and a child network `FC`. At line 37, `Net` calls `FC`, thus on the interface the corresponding tensor state box populates a clickable arrow. When the programmer clicks it, the whole live visualization goes into the "deeper" network `FC` and shows the tensor state boxes of the range between line 14 and line 18. The back arrow at the screen center enables back ways to the

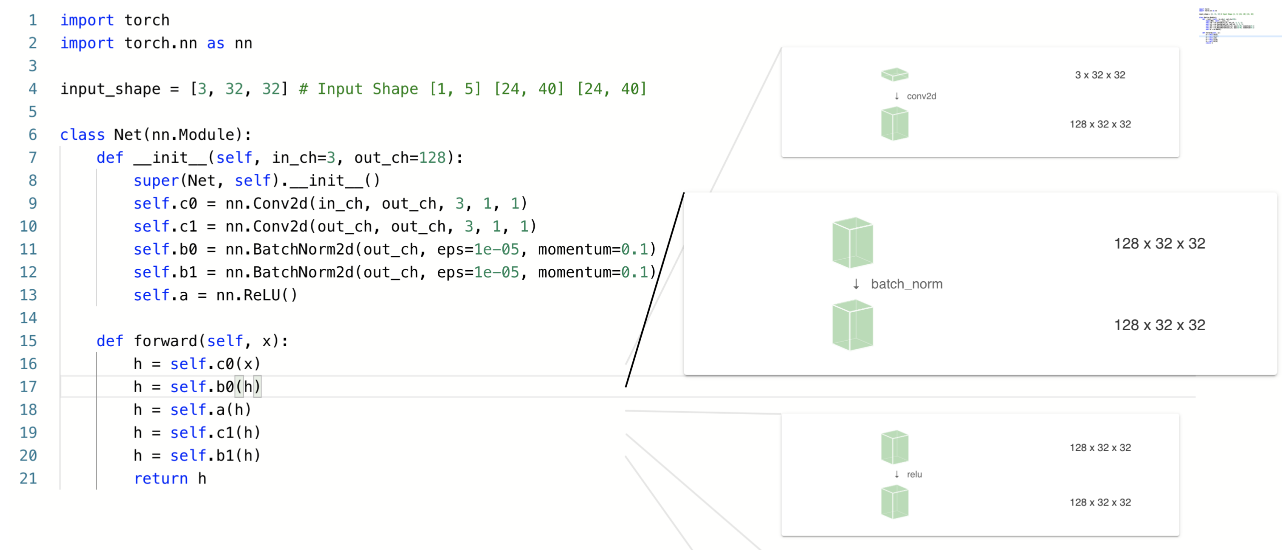


Figure 1: The visualization of tensor state box.

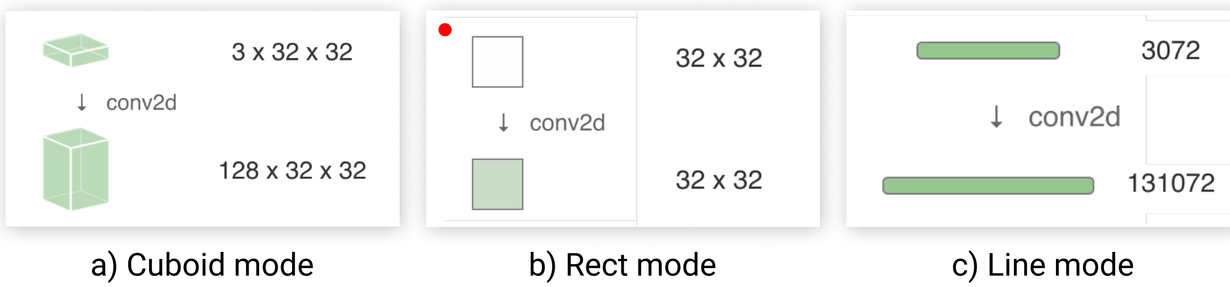


Figure 2: The user can switch the tensor state box display within the three view modes.

parent network. The user can also navigate between the parent and the child networks using a breadcrumb widget at the top-left of the code editor.

5 IMPLEMENTATION OF ODEN AS AN IDE

NN architecture editing is not a standalone process in DL development. Programmers repeat the process when they want to try new ideas in their network architecture to pursue better model performance[33]. Therefore, we package the live NN architecture editing with some additional functions, including experimentation, into an IDE named ODEN to make the live NN architecture editing speak of its value in a repetitive "editing then experimenting" process. This section briefly describes how we implemented the prototype of ODEN. We first overview the user interface, then walk through the functions and the system implementation details.

5.1 User Interface

The user interface is shown in Figure 4. Figure 4 (a) is the live NN architecture editor described in the section 4. ① is the header bar that contains some configurable items (including the dropdown list to configure the visualization view mode); ② and ③ are the code editor and the live network architecture visualization; ④ is a bidirectional manipulable widget where the programmer can drag the sliders to mutate the input tensor shape; ⑤ is a drawer that contains the experimentation panel where the programmer can instantly test their network structures. The experimentation panel pops up from the bottom when the drawer is clicked, and the user interface will become the Figure 4 (b). In the experimentation panel, ⑥ is the editor to write programs that execute experiments on the network architectures defined in ② and ③. ⑦ is a visualization for the experiment version control, and ⑧ is a parallel coordinate plot that dynamically tracks hyper-parameters, network architectures, and objective values.

5.2 Additional Features

5.2.1 Bidirectional manipulable widget. In the current ODEN prototype, to enable the live network visualization, the programmer needs to assign a tuple of the input tensor shape to a variable `input_shape`. *ODEN* defines a special comment notation, `# Input Shape:`, that populates a slider widget for input tensor shape manipulation. The widget provides slider interfaces to

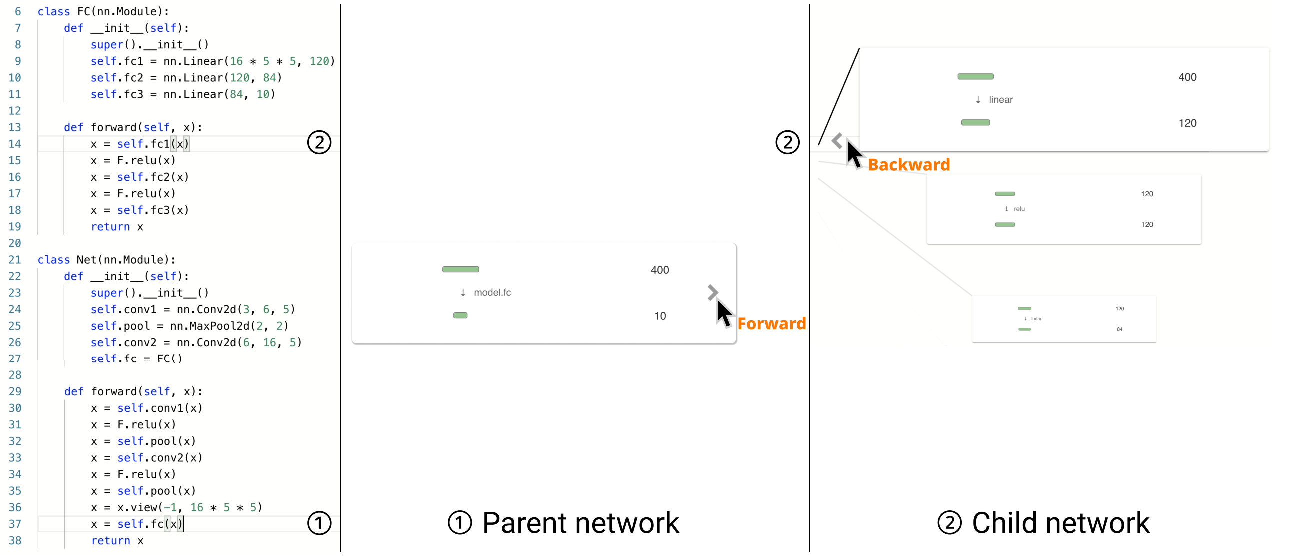


Figure 3: Nested network architecture inspection. The user can click on arrows in the tensor box visualizations in the parent network to expand child network details. The arrow between the editor and the visualization then enables the user to go back to a "shallower" network level.

assist intuitive manipulation of the input shape. When manipulated, the widget reflects the change in the text editor by updating the tensor shape value. Inversely, value change of `input_shape` also reflects on the widget synchronously. The programmer can attach this particular comment following with the slider range to the line where `input_shape` is assigned (See Listing 2). Then the slider widget instantly populates at the top of the live visualization (Figure 4 (a) ④).

Listing 2: Comments to populate bidirectional manipulatable hyper-parameter axes.

```
input_shape = [3, 30, 32] # Input Shape: [1, 32] [4, 128]
    ↪ [4, 128]
```

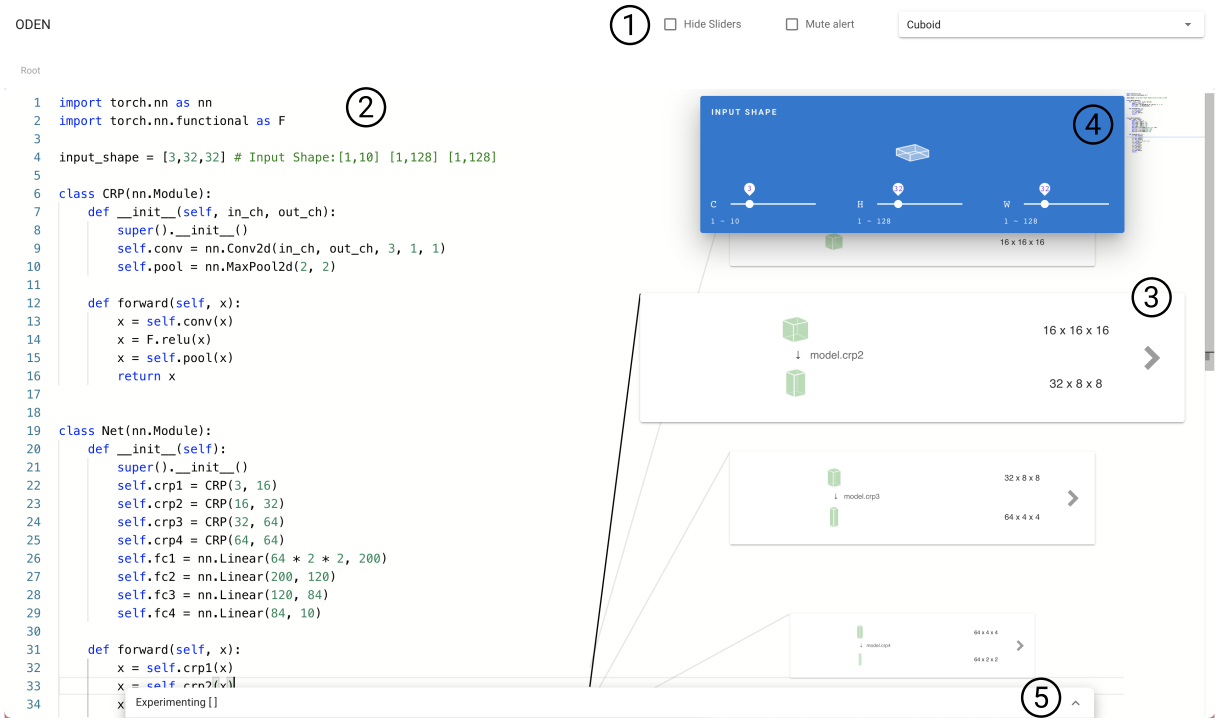
5.2.2 The code editor in the experimentation panel. There is another standard code editor (Figure 4 (b) ⑤) in the experimentation panel. In the current implementation, the programmer must define four providers in the editor before executing experiments, including a data loader provider, an optimizer and criterion provider, a training provider, and a testing provider. The programmer needs to append a special comment notation `# HyperOpt:` to the lines that contain hyper-parameter values in this code editor. Listing 3 shows the example. In the notation, the brackets and parentheses define the tweaking range and the data type of these hyper-parameters. When the notation is appended, the experiment version control and the parallel coordinate plot will responsively record a new hyper-parameter in each experiment.

Listing 3: Comments to populate bidirectional manipulatable hyper-parameter axes.

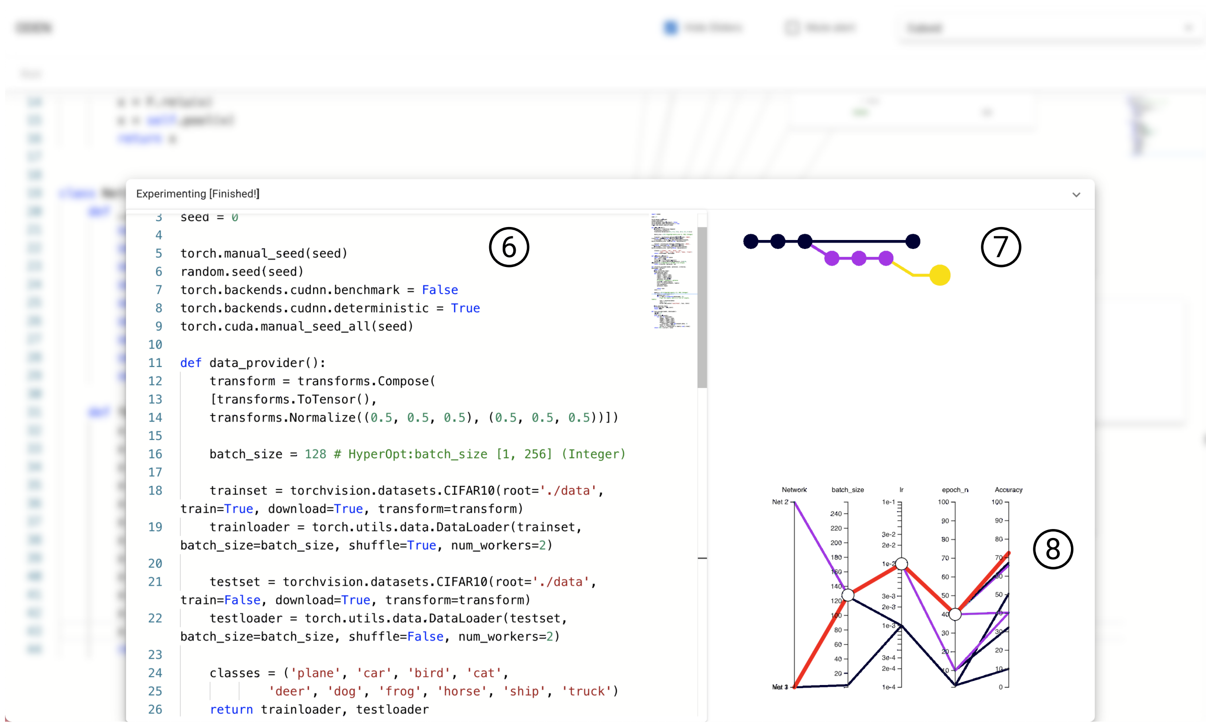
```
batch_size = 63 # HyperOpt:batch_size [1, 256] (Integer)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum
    ↪ =0.9) # HyperOpt:lr [0.0001, 0.1] (Float)
```

5.2.3 Experiment version control visualization. Inspired by version control systems like Git, *ODEN* manages and visualizes the experiments using a tree structure (See Figure 5 (a)). *ODEN* watches the modification of all denoted hyper-parameters and the whole network architecture, then synchronously populates new experiment dots in the visualization. Experiment dots in the same branch share the same NN architecture. Suppose the programmer goes back to the live NN architecture editor and modifies the network architecture. In this case, the visualization will instantly populate a new branch and a new hollow dot that represents the current in-edit experiment. If the programmer mutates the hyper-parameter values denoted by the particular comment, the visualization will also populate a new hollow dot but without switching the branch. All experiment dots are clickable: the solid dots serve as triggers to restore the codebase of previous experiments, and the hollow one is for executing the in-edit experiment. Once the in-edit experiment dot is clicked, the experiment will start training, then testing, and finally render the new experiment result in the parallel coordinate plot.

5.2.4 Parallel coordinate plot for checking hyper-parameters and objective values. Parallel coordinate plots (PCPs) have been developed in hyper-parameter optimization analytic tools [19] and have become an established practice in representing DL experiments. Similarly, *ODEN* utilizes a parallel coordinate plot (PCP) to draw the multi-dimensional experiment data consisting of hyper-parameters and experiment result (In the example, it is `Accuracy`. See Figure 5 (b)). When the programmer adds a new comment to the hyper-parameter to tweak and watch, the visualization instantly populates a new draggable axis in the PCP. The hyper-parameter axis acts like a slider that the programmer can directly manipulate the parameter value in the visualization. And vice versa, the visualization is bidirectional, which means that if the programmer edits



(a) Live neural network architecture editor



(b) Expanded experimentation panel

Figure 4: The user interface of ODEN. (1) header bar contains some configurable items; (2) code editor for live NN architecture editing; (3) live NN architecture visualization; (4) bidirectional manipulable widget to mutate the input tensor shape; (5) clickable drawer that contains the experimentation panel; (6) editor to define experimentation providers; (7) experiment version control visualization; (8) bidirectional manipulable parallel coordinate plot that tracks hyper-parameters and objective values.

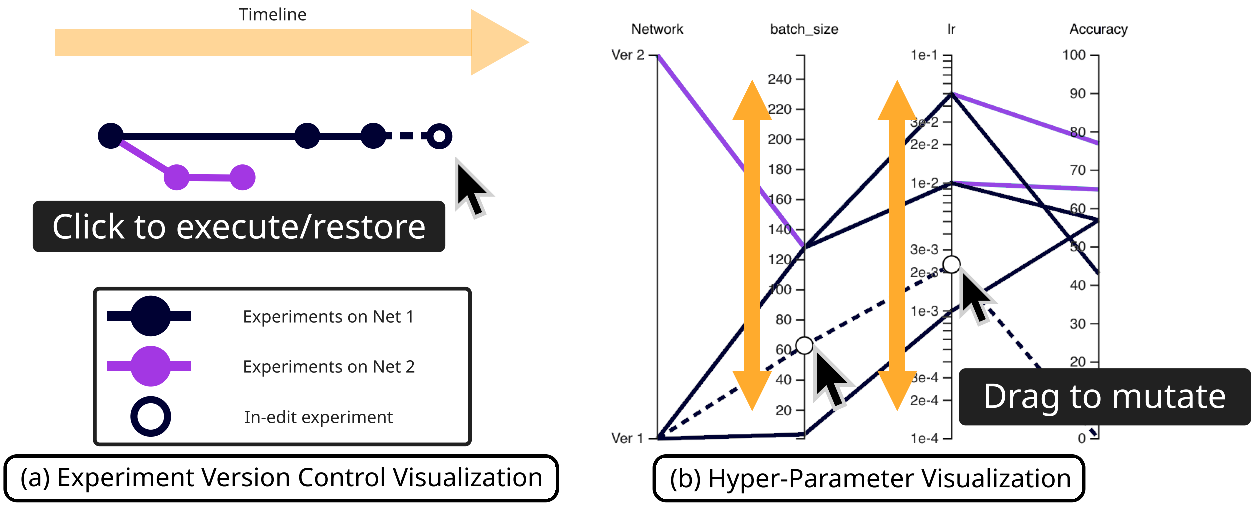


Figure 5: Two visualizations in the experimentation panel.

the parameter value in the text editor, the value will also instantly reflect on the visualization.

5.3 Implementation Details

The current prototype of ODEN is a web-based application consisting of a Python web server and an HTML/Javascript client. The Python web server instantiates a GPU environment, injects a PyTorch analyzer into the environment, and constantly collects the information required for live visualization. The analysis of the Python server is based on actual program execution rather than statical code check like [15]. According to the discussion in subsection 3.3, tensor shape mismatch in linear layers can be addressed by static code check because the layer definition directly indicates the input and output tensor shape. However, the output tensor shape of layers like convolution layers and pooling layers is influenced by the layer definition and the input tensor shape, which makes static code difficult to cover all layer types in NN architecture. Although the analyzer also supports a CPU-only environment, we recommend setting up the server on a GPU-on machine because the training process relies highly on GPU acceleration. The client is built in Electron and Vue.js and utilizes Monaco Editor as the text code editor in ODEN. A user can install the client Electron app on a laptop while the Python server runs on a GPU server as a typical use case. The communication between the client and the server is based on the JSON-RPC protocol. When the user edits the network program or clicks on the hollow dot that controls the experiment, the code is sent to the server and saved as a separate codebase file with a hashed filename.

Network Tracking Session. We simulate a single iteration of the user-defined network forward and backward computation in the network tracking session to track the tensor shape transformation in the whole network. The user must define a class named `Network` and a list named `input_shape` to enable the live inspection. The server-side analyzer loads the `Network` class definition and reads the `input_shape` list as the input tensor shape. The analyzer then starts a tracking session, creates a dummy input tensor that is in the same shape as `input_shape`, instantiates the user network, and executes a simple iteration. Before that, the session will inject a hook function to all PyTorch methods and modules in the computation graph, which records the input and output `torch.Tensor` instance and gets their shape by calling built-in `size()` method of the instance.

Code Association and Hierarchical Tensor Shape. The tracking session utilizes a stack trace to record trace frames every time the hooked computation nodes are invoked. The session first tracks tensor shapes together with frame information while retaining the call stack depth information. It then aggregates it into a nested form that aligns with where the module or method is invoked in the user code. For those nested networks that several subnetwork modules are defined and called by each other, the shape information will also have a nested representation.

Extensibility of Experiment Mode. In the experiment mode, the user is required to implement four providers (See Figure 4 ⑥). The data loader provider should return a training data loader and a test data loader; the optimizer and criterion provider should return an optimizer, a criterion layer, and a network instance; the training provider should receive a network instance, an optimizer, a criterion layer, and a training data loader, then return a trained model; a testing provider should receive a trained model and a test data loader, then return the test metric to display and record. When the providers work as intended and the pipeline works as a whole, the user can start the experiment. Nevertheless, the user can customize these providers to extend to other loss functions or data loaders.

6 USER STUDY

We conducted an exploratory study to obtain novice user feedback on ODEN. The main goal of the study is to investigate the usability, the limitations, and the potential of the live neural network architecture editor. Seven programmers participated in the study. In our pre-experiment questionnaire, the participants generally identify themselves as experienced in general-purpose programming while not familiar with DL development. We requested them to go through the usage scenario described in the PyTorch official tutorial, "Deep Learning with PyTorch: A 60 Minute Blitz, Training a Classifier," copy-paste code snippets into the tool, and iteratively adjust the parameters and the network structure in the program until satisfied. They were asked to compare their impression of using our tool with their prior experience of DL applications development using other tools.

6.1 Participants

Seven participants aged from 23 to 32 years old (mean 26.57) took part in the experiment, including five males and two females. All of them are programmers (experience from 3 to 9 years; mean 5.14 and standard deviation 1.95), and they are learning DL programming to utilize the techniques for various purposes (e.g., research projects and Kaggle contest). They have average knowledge towards DL theory and know how DL works (for 5-point Likert scale of the question "How much do you know about deep learning theory?", mean 3 and standard deviation 1), but lack knowledge about the DL programming practice and do not develop DL models very often (for 5-point Likert scale of the question "How much do you know about deep learning programming paradigm?", mean 2.71 and standard deviation 0.76; for 5-point Likert scale of the question "How often do you develop a deep model?", mean 2.14 and standard deviation 0.90). In terms of the developing tools for DL, two participants (P1 and P5) who mainly use existing TensorFlow models released on GitHub and slightly edit the program prefer VSCode to fit their needs. Three participants (P3, P4, and P7) are PyTorch users, who develop their models with VSCode and notebook programming tools like Jupyter and Google Colab. For detailed information of the participants, please refer to Table 1.

6.2 Experimental Setup

The instructor set up the Electron client on a MacBook Pro 16 inch 2019 with a 1792×1120 equivalent resolution. The Python server ran on a Ubuntu 20.04 machine with an RTX 2080 Super GPU, and the server and the client laptop were in the same private network. All the participants used remote desktop software to connect to the client laptop, then remotely controlled and operated in the client interface; meanwhile, the instructor and the participant joined the same online meeting. The user interface was in full-screen mode during the study. Next, the experiment follows the steps described below, and the whole experiment took about 1.5 to 2 hours for each participant.

6.2.1 Pre-experiment questionnaire. Each participant was required to fill out an online form that collects their daily programming language, general-purpose programming skill level, experience in DL programming, and ML knowledge level.

6.2.2 Introduction, tutorial, and practice. The instructor gave a 5-15 minute introduction and tutorial about ODEN IDE. First, the instructor went through the PyTorch tutorial with the participant and introduced the related concepts. Next, the instructor demonstrated the live NN architecture editing and other interactive functions in ODEN. Finally, the participant could practice the system operations as they want.

6.2.3 Network editing and experimentation iteration. The participants were required to finish an open-ended task within 1.5 hours maximum: use ODEN to iterate the network and experimentation programs until they are satisfied with the final model accuracy based on the programs in the "Deep Learning with PyTorch: A 60 Minute Blitz, Training a Classifier" tutorial[2]. The participants start from code snippets from the tutorial and then refine them. In each iteration to improve the model, the participant needs to edit the network structure or tune the hyper-parameters, then train and test the model. The initial network in the tutorial is LeNet [16], the dataset to train and test on is CIFAR10, and the model target is to classify images in ten classes of CIFAR10. The initial experiment setup from the tutorial produces an accuracy of 55.83%, which is much better than random guessing (10%) and not sufficient compared to state-of-the-art (accuracy over 90%). During the open-ended task, the participant was free to ask the instructor about the suggestions to improve the accuracy, search NN architecture design tricks on Google, and refer to classic NN architectures and open-source code. In addition, we encourage participants to think aloud during the task. The participant can also stop the study early if they are satisfied with the experiment results.

6.2.4 Post-experiment questionnaire. Each participant was asked to fill out a post-experiment form. The form includes a standard system usability scale [6] (ten questions) to verify the usefulness of the live NN architecture editor, one question for rating the transition between the editing and experimentation, one question for rating the nested network inspection, and two free comment questions about the good and bad points of the user interface. When the participants filled out the questionnaire, we had a casual interview with them to talk about their impression of ODEN.

6.3 Results and Lessons Learned

The results of the improved model accuracy are shown in Figure 6. From P1 to P7, the participants conducted 6, 8, 3, 6, 6, 11, and 3 iterations to improve the model accuracy. Compared with the model accuracy of the default example code from the tutorial (= 55.83%), the participants improved the model accuracy to 75.7%, 67.6%, 81.0%, 61.0%, 73.0%, 67.4%, and 76.1% within 1.5 hours (for each participant's best record). All participants succeeded in increasing the model accuracy. The analysis of the model accuracy using paired t-test showed that the final model accuracy during the task was significantly different ($p < 0.05$) from the initial model accuracy. The results of the post-questionnaire consist of the mean, standard deviation, and the percentage of positive responses (>3 on a 5-point Likert scale).Table 2 shows the results. The final SUS score is calculated by averaging each participant's SUS score, and

[2] Deep Learning with PyTorch: A 60 Minute Blitz, Training a Classifier, visited at Sep. 6th, 2021

Table 1: The basic information of seven participants, including their occupation, preferred programming languages, purpose to learn DL, and where they learned DL.

P#	Occupation	PL	Purpose	Where to learn DL
P1	Student	Java	Research	Textbooks & Online Courses
P2	Student	Javascript	Research	Lectures
P3	Student	Python	Research	Lectures
P4	Student	Python	Research	Online Courses & Library documents
P5	Student	Python	Research	Lectures
P6	Software engineer	Python & Javascript	Kaggle	Online Courses
P7	Student	Python	Research	Lectures & Papers

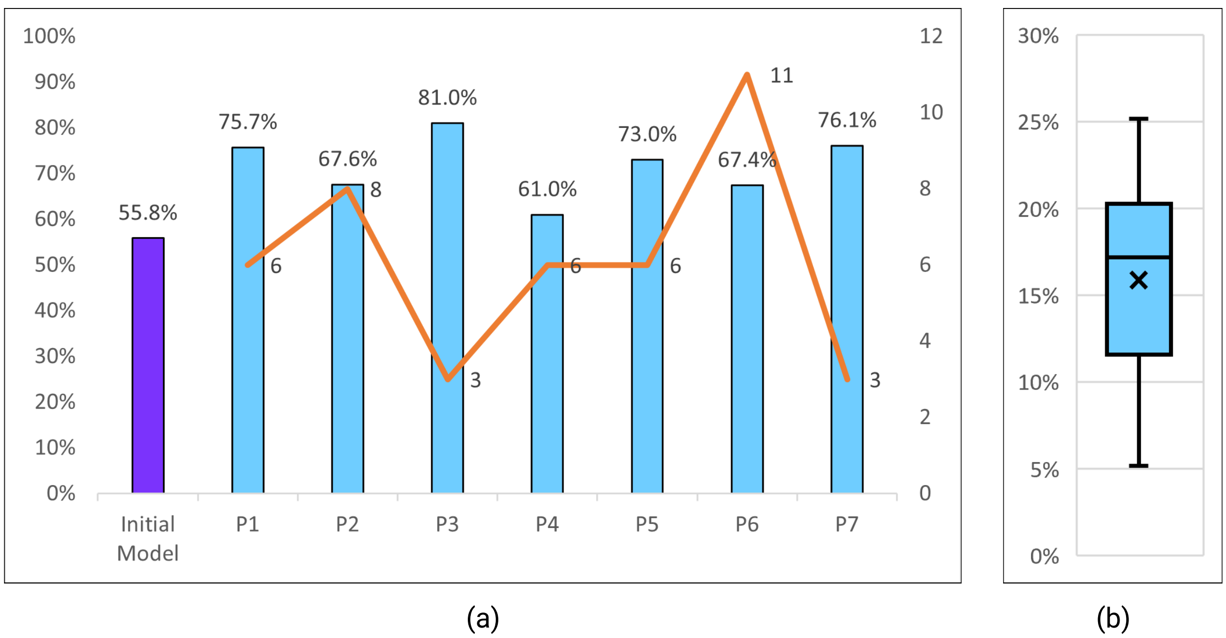


Figure 6: Results of the improved model accuracy and experiment iteration numbers. (a) Left y-axis: Accuracy of the initial model and the best records of all participants during the user study; right y-axis: the experiment iteration numbers by participants. (b) Box chart of the incremental accuracy.

the details of the SUS score calculation method are described in [6]. The final SUS score of the live NN architecture editor is 84.64. (The standard SUS score is 68, and our score of 84.64 is regarded as excellent and has the grading scale of Grade A [5].

6.3.1 Several usage scenarios exist in actual NN architecture editing. . We found that during the open-ended task phase, the participants might choose three different usage scenarios to refine the NN architecture: the first one is to add or reduce layers slightly on the example network architecture following some heuristics (P1, P4 "*I know the common way to improve the model accuracy is to make the network deeper by adding more layers to it*"); the second one is to search for more complicated and classic NN architecture for the same task and modify the original network by making it similar to the reference (P3 "*I take the VGG [25] network architecture as the reference and use the similar shape and layer sequences in my network.*"); the third one is to program a same complicated network such as VGG in the editor completely, or even copy-paste a network architecture program into the live NN architecture editor then modify (P7 "*I'd like to see how the ResNet [10] works on the dataset. I will copy-paste an existing program from GitHub into the interface.*"). In the conversation with the participants, we knew which usage scenario to choose is case by case. (P4 "For me, to choose to program from scratch or copy-pasting depends on how casual the model building is.") In the three usage scenarios, participants successfully improved the model accuracy within the limited time using ODEN, and the live NN architecture visualization smoothly visualizes all the programs. In practical NN architecture building, users will follow some scenarios that are similar to the strategies our participants adopted in our user study: users may build a NN from scratch by referring to a diagram in a research paper; or users

Table 2: Results of the post-experiment questionnaire. $\Uparrow$ indicates higher scores are better. $\Downarrow$ for the other case.

#	Questions	Mean	SD	%
1	I think that I would like to use the live network editing frequently. $\Uparrow$	4.71	0.49	7/7
2	I found the live network editing unnecessarily complex. $\Downarrow$	2.14	0.90	1/7
3	I thought the live network editing was easy to use. $\Uparrow$	4.43	0.53	7/7
4	I think that I would need the support of a technical person to be able to use this live network editing. $\Downarrow$	2.14	1.07	2/7
5	I found the various functions in this network editing were well integrated. $\Uparrow$	4.29	0.49	7/7
6	I thought there was too much inconsistency in the live network editing. $\Downarrow$	1.57	0.53	0/7
7	I would imagine that most people would learn to use the live network editing very quickly. $\Uparrow$	4.57	0.53	7/7
8	I found the live network editing very cumbersome to use. $\Downarrow$	1.14	0.38	0/7
9	I felt very confident using the live network editing. $\Uparrow$	4.43	0.53	7/7
10	I needed to learn a lot of things before I could get going with the live network editing. $\Downarrow$	1.57	0.53	0/7
11	Rates of the transition between the editing and experimentation. $\Uparrow$	4.29	0.49	7/7
12	Rates of the inspection support for nested networks. $\Uparrow$	4.43	0.53	7/7

may start from copy-and-pasting existing programs then modify them. We believe that ODEN can support either actual scenarios above.

6.3.2 Instant synchronization between the network program and the visualization greatly assists in NN architecture editing. . In all of the usage scenarios above, the participants appreciated the live programming nature in the live NN architecture editor. P3 commented that "*I also often encountered the shape mismatch errors in my implementing experience. They indeed bother me. With ODEN's live NN architecture editor, I can significantly avoid these errors if I face not only the text program but also a visualization like this.*" P4 appreciated that "*When I want to implement a network architecture, I usually have to draw a scratch of the tensor shape sequence on a whiteboard as a reference. The live NN architecture editor benefits me a lot because, with it, I won't need to draw the network from scratch anymore.*" P1 and P4 said, "*I will have a rough diagram towards the network in my mind (before the implementation). With the visualization, I can verify whether the code is consistent with my mental image or not.*" P7 rethought her programming experience using Google Colab and commented: "*When working on neural networks with PyTorch, I often get confused with shape inconsistency errors. I'd love to see some live shape visualization implemented in my work environment as well.*"

6.3.3 Support for nested network architectures enables better code reuse. During the task and the interview, some participants found the nested network support greatly helps them reuse existing NN programs. "*I seldom develop my own neural network architecture from scratch. On the contrary, I often start from cloning an existing GitHub repository with trained neural networks.*" said P1, "*When I conduct transfer learning on my own dataset, I sometimes need to adjust the NN architecture depending on my case. The support for nested networks helps a lot for me because most of the open-source neural network architectures are not linear but hierarchical.*" P3 appreciated that "*I can directly copy and paste my previous networks into this interface!*"

6.3.4 Smooth transition between NN architecture editing and experimentation. All of our participants thought the whole exploration process in the open-ended task was exciting and relaxing and spoke high of the transition between NN architecture editing and experimentation. "*Because playing with the experimentation won't cause stress to me.*" P5 said, "*The tracked overall experiment history helps me record the accuracy right after my editing on the neural network architecture.*" P1 appreciated the special notation to specify hyper-parameters and the experiment version visualization of the experimentation panel: "*It (the* `# HyperOpt:` *notation) makes every variable clear. I can easily figure out which hyper-parameter matters in experimentation.*"

6.3.5 Non-positive comments on ODEN implementation. Participants provided some negative comments about the implementation of ODEN.

- The live visualization is a bit intrusive – P3 "*Although it (the live visualization) helps me a lot, it contains too much blank and occupies too much space on the screen.*" P7 "*I thought the live visualization could be improved by having visualization for each line in-situ. It was a bit hard to navigate which shape corresponds to which line in the current implementation. I hope the shape visualization could be more static.*" As a quick fix to this feedback, we implemented another two configurable modes in ODEN: a "calm mode" that disables the visualizations but decorates the code editor background with text-form tensor shape; a "stealth mode" that only retains the original code editor.
- How to design a neural network architecture is puzzling to novices – "*Without your (the instructor) advice on how to modify the network, I may have no idea what I should do to*

improve the performance. Therefore I hope that a system can guide how to design a neural network architecture."

- Implementation limitation – P1 encountered an implementation bug when copying and pasting a ResNet18 architecture into ODEN from a GitHub repository. For code that calls several layers in the same line but does not wrap these layers into a child network, e.g., `F.relu(self.batchnorm(self.conv(x)))`, ODEN did not provide the best visualization layout in the corresponding box visualization and resulted in text overlapping. We then fixed the bug after the user study.

7 DISCUSSION AND LIMITATIONS

The choice to build a custom IDE and the extensibility for common editors. The current implementation of *ODEN* is merely a research prototype. We built a custom IDE because it is easier to implement and test our live visualization and editing features on a custom IDE than on existing ones. The major challenge of integrating all our proposed features into an existing IDE is the significant engineering effort needed to hack into a full-fledged system. On the other hand, the advantage of building an IDE from scratch is that we can fully utilize the flexibility of engineering and maximize the liveness and seamlessness of our live editor. Therefore, we focused on our core contributions and left the integration to other existing IDEs as future work. Nevertheless, our standalone server-side analyzer retains the extensibility for other clients. We believe it is possible to implement the frontend client as a plugin for existing IDEs meanwhile reusing the server-side analyzer.

Versatility on existing neural network architectures. Current ODEN implementation only supports CNN architectures that receive 2D images as input. We have tested ODEN on nested network architectures like DenseNet [13], VGG [25], ResNet [11], and DCGAN [23], inputting with code from public GitHub repositories. ODEN successfully analyzed and visualized these architectures. We think ODEN is versatile enough to support the typical usage scenario that the user directly reuses a network architecture from a public codebase. We have conducted another casual expert interview involving four DL experts after our implementation. We showed the test results to the experts. They gave positive feedback and agreed that ODEN can conveniently visualize their existing codebases.

Automatic tensor shape mismatch fix. The current design of the live NN architecture editor provides synchronous tensor shape visualization for the programmer's reference. The programmer still needs to manually fix the tensor shape mismatch in the program. The automatic tensor shape mismatch fix is attractive to novice programmers. Still, since most layers receive a specific tensor shape pattern rather than a unique shape, there is no unique solution to a particular tensor shape mismatch. Instead, automatically suggesting several solution candidates can be future work to this problem.

Guidance for neural network architecture design. One of our participants during the user study stated that novice DL programmers need advice and heuristics about how to design a neural network architecture with good performance. Yan et al. created a large-scale visualization to help find insights of design choice in NN architecture [33], and Schoop et al. designed a system that analyzes a DL program and advises on potential performance issues based on heuristics [24]. How an interface can dynamically predict the performance of an in-edit NN architecture and suggest novice programmers better design choices in NN architecture building remains to be explored.

The potential application for model migration. Model and codebase migration between different DL libraries require much engineering effort. Although projects such as ONNX [4] provide a standard representation for models and networks in various libraries, a user interface still lacks that supports model migration between DL frameworks, e.g., model weight matching, codebase conversion, and accuracy validation. Since our current prototype only supports PyTorch, one of our future works is to design a live programming interface that supports model migration.

Further enhancement for parameter tweaking. The DL parameter tweaking [12] is challenging even in sandboxed experiments. During the interview, one of our participants said: "Sometimes it is just like lotteries". The proposed visualization and IDE mainly provide a stage for the live editor. How to extend the visualization to support parameter tweaking remains to be explored.

8 CONCLUSION

We proposed a live NN architecture editor that utilizes live programming techniques to help novice DL programmers solve tensor shape mismatch errors in NN architecture editing. The editor synchronously displays tensor box visualizations that show the input/output tensor shapes at lines that the tensor computation is executed. We implemented the live editor and integrated it into an IDE called *ODEN*. With ODEN, programmers can freely edit the neural network architecture and instantly train/test the edited network architecture. We conducted an exploratory user study. We found that the live NN architecture editor is intuitive and helps solve tensor shape mismatch errors significantly. We found some practical usage scenarios during the interviews with our participants. Further enhanced interface for hyper-parameter tweaking and NN architecture design is the future work.

ACKNOWLEDGMENTS

This work was supported by JST CREST Grant Number JPMJCR17A1, JPMJCR20D4, and JSPS KAKENHI Grant Number JP19K20316.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, ..., and Xiaoqiang Zheng. 2015. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, USA, 265–283.

[2] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, et al. 2019. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. In *Proceedings of the 2nd Systems and Machine Learning Conference (SysML)*. Palo Alto, CA, USA, 88:1–88:12. https://mlsys.org/Conferences/2019/doc/2019/88.pdf

[3] Moayad Alshangiti, Hitesh Sapkota, Pradeep Murukannaiah, Xumin Liu, and Qi Yu. 2019. Why is Developing Machine Learning Applications Challenging? A Study on Stack Overflow Posts. In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Porto de Galinhas, Brazil, 1–11. https://doi.org/10.1109/ESEM.2019.8870187

[4] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[5] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *J. Usability Studies* 4, 3 (May 2009), 114–123.

[6] John Brooke. 1996. SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry* (0 ed.). CRC Press, Boca Raton, FL, USA, 207–212. https://doi.org/10.1201/9781498710411-35

[7] Carrie J Cai and Philip J Guo. 2019. Software Developers Learning Machine Learning: Motivations, Hurdles, and Desires. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Memphis, TN, USA, 25–34. https://doi.org/10.1109/VLHCC.2019.8818751

[8] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE)* (Denver, Colorado, USA). ACM, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[9] Christopher Michael Hancock. 2003. Real-time programming and the big ideas of computational literacy. *Thesis (Ph. D.)–Massachusetts Institute of Technology, School of Architecture and Planning, Program in Media Arts and Sciences* (2003), 1–121. http://hdl.handle.net/1721.1/61549

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 770–778.

[12] Keita Higuchi, Shotaro Sano, and Takeo Igarashi. 2021. Interactive Hyperparameter Optimization with Paintable Timelines. In *Proceedings of the 2021 Designing Interactive Systems Conference (DIS '21)*. ACM, New York, NY, USA, 1518–1528. https://doi.org/10.1145/3461778.3462077

[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '17)*. IEEE, 4700–4708.

[14] Jun Kato and Masataka Goto. 2017. F3.Js: A Parametric Design Tool for Physical Computing Devices for Both Interaction Designers and End-users. In *Proceedings of the 2017 Conference on Designing Interactive Systems (DIS '17)* (Edinburgh, United Kingdom). ACM, New York, NY, USA, 1099–1110. https://doi.org/10.1145/3064663.3064681

[15] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static analysis of shape in TensorFlow programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[17] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[18] Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. 2017. Live Data Structure Programming. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17)* (Brussels, Belgium). ACM, New York, NY, USA, Article 26, 7 pages. https://doi.org/10.1145/3079368.3079400

[19] Heungseok Park, Yoonsoo Nam, Ji-Hoon Kim, and Jaegul Choo. 2021. HyperTendril: Visual Analytics for User-Driven Hyperparameter Optimization of Deep Neural Networks. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 1407–1416. https://doi.org/10.1109/TVCG.2020.3030380

[20] Adam Paszke, Sam Gross, Francisco Massa, ..., and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of Advances in Neural Information Processing Systems 32 (NeurIPS)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., Vancouver, Canada, 8024–8035.

[21] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. 2010. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology* (New York, New York, USA) *(UIST '10)*. Association for Computing Machinery, New York, NY, USA, 37–46.

[22] Kayur Patel, James Fogarty, James A. Landay, and Beverly Harrison. 2008. Investigating Statistical Machine Learning as a Tool for Software Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)* (Florence, Italy). ACM, New York, NY, USA, 667–676. https://doi.org/10.1145/1357054.1357160

[23] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR* abs/1511.06434 (2016).

[24] Eldon Schoop, Forrest Huang, and Bjoern Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs Using Program Structure and Model Behavior. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 310, 16 pages. https://doi.org/10.1145/3411764.3445538

[25] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of International Conference on Learning Representations (ICLR)*. San Diego, CA, USA, 1–14. http://arxiv.org/abs/1409.1556

[26] Daniel Smilkov, Shan Carter, D Sculley, Fernanda B Viégas, and Martin Wattenberg. 2016. Direct-manipulation visualization of deep networks. In *Proceedings of the ACM SIGKDD Workshop on Interactive Data Exploration and Analytics (IDEA '16)*. San Francisco, CA, USA, 115–119. http://poloclub.gatech.edu/idea2016/papers/p115-smilkov.pdf

[27] Sony Network Communications Inc. 2018. Neural Network Console. https://dl.sony.com.

[28] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. 2018. LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 667–676. https://doi.org/10.1109/TVCG.2017.2744158

[29] Steven L Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE)*. IEEE, San Francisco, CA, USA, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[30] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, and Hiroyuki ... Yamazaki Vincent. 2019. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, New York, NY, USA, 2002–2011. https://doi.org/10.1145/3292500.3330756

[31] Jeremy Warner and Philip J. Guo. 2017. CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1136–1141. https://doi.org/10.1145/3025453.3025876

[32] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. 2017. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2017), 1–12. https://doi.org/10.1109/TVCG.2017.2744878

[33] Litao Yan, Elena L. Glassman, and Tianyi Zhang. 2021. Visualizing Examples of Deep Neural Networks at Scale. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3411764.3445654

[34] Qian Yang, Jina Suh, Nan-Chen Chen, and Gonzalo Ramos. 2018. Grounding Interactive Machine Learning Tool Design in How Non-Experts Actually Build Models. In *Proceedings of the 2018 Designing Interactive Systems Conference (DIS '18)* (Hong Kong, China). ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/3196709.3196729

[35] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd ACM Symposium on User Interface Software and Technology (UIST'20)*. ACM, New York, NY, USA, 126–139. https://doi.org/10.1145/3379337.3415890

[36] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *Proceedings of IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Berlin, Germany, 104–115. https://doi.org/10.1109/ISSRE.2019.00020