Explore Your Network in Minutes: A Rapid Prototyping Toolkit for Understanding Neural Networks with Visual Analytics

Shaoxuan Lai, Wanna Luan, and Jun Tao, Member, IEEE

Abstract—Neural networks attract significant attention in almost every field due to their widespread applications in various tasks. However, developers often struggle with debugging due to the black-box nature of neural networks. Visual analytics provides an intuitive way for developers to understand the hidden states and underlying complex transformations in neural networks. Existing visual analytics tools for neural networks have been demonstrated to be effective in providing useful hints for debugging certain network architectures. However, these approaches are often architecture-specific with strong assumptions of how the network should be understood. This limits their use when the network architecture or the exploration goal changes. In this paper, we present a general model and a programming toolkit, Neural Network Visualization Builder (NNVisBuilder), for prototyping visual analytics systems to understand neural networks. NNVisBuilder covers the common data transformation and interaction model involved in existing tools for exploring neural networks. It enables developers to customize a visual analytics interface for answering their specific questions about networks. NNVisBuilder is compatible with PyTorch so that developers can integrate the visualization code into their learning code seamlessly. We demonstrate the applicability by reproducing several existing visual analytics systems for networks with NNVisBuilder. The source code and some example cases can be found at https://github.com/sysuvis/NVB.

Index Terms—Visualization model, toolkit, neural networks

1 Introduction

In recent years, neural networks have achieved tremendous success in fields such as image recognition, natural language processing, speech recognition, and game strategy. However, the black-box nature of neural networks has brought some difficulties to their exploration and application. The black-box nature refers to the opacity of internal workings within neural networks, making it difficult for people to understand how neural networks learn from data to make predictions and decisions. To understand the internal workings, various methods have been proposed, among which visualization is relatively intuitive and effective. The visualization technique helps to display the information and mechanism of neural networks, offering insights into learned features and patterns, thus further promoting their application in diverse fields [1, 12, 17].

Existing approaches may reveal the behavior of neural networks by showing distinct types of information. Some approaches visualize information of the network itself, such as the network architecture [25, 31] and the parameters of networks [36]. Some works reveal the behavior by showing how the neural network responds to data samples. These approaches visualize the (intermediate) output of neural networks, such as activation [36, 44] and embedding space [37, 38]. They may compare samples to identify the ones leading to anomalous behavior of the network [22]. Sophisticated approaches are developed to derive attributes from the neural network for understanding how the network transforms input data at different levels, such as activation maximization [9, 35, 49]. These approaches can be considered feature generation techniques.

However, the aforementioned approaches all analyze neural networks based on certain assumptions, and they may not be able to answer other questions. For example, the approaches visualizing network architecture or training data may not reveal the internal information of the network, which is crucial for understanding the internal workings of the neural network. And, visual analytics approaches based on internal information (e.g., activations), network parameters, and derived features usually target specific network architectures [1] and are difficult to customize and extend. In addition, building visual analytics interfaces is costly, as the complex data processing and interaction scheme need to be implemented by the users. This can be challenging for non-visualization experts.

To tackle these challenges, we design NNVisBuilder, a programming toolkit for building prototypes of customized visual analytics interfaces. It can be applied to neural networks of various architectures and answer diverse kinds of questions. NNVisBuilder adopts the client-server model. The server is responsible for data management and interaction handling. It is compatible with PyTorch so that users can easily integrate NNVisBuilder into their learning code seamlessly. The client is developed using D3 [3] to exploit its rich visualization resources. Besides, we have proposed a model for abstracting the data and workflow of visual analytics interfaces for neural networks. This model facilitates effective analysis of analytics systems for neural networks and assists non-visualization experts in designing interactive systems.

The major contributions of this work are as follows:

- We design a programming toolkit, NNVisBuilder, to help developers rapidly build prototypes of customized interactive visual analytics interfaces for understanding neural networks.
- We propose a data processing and interaction workflow in visual analytics for neural networks based on the dimensions of tensors, allowing users to build interfaces at a higher level.
- We introduce a model to abstract the visual analytics interface for neural networks into flowcharts. This helps visually summarize and compare analytics interfaces. It also guides the design and implementation of interfaces using NNVisBuilder.

2 Related Work

Our work falls into the category of Vis4AI (Visualization for Artificial Intelligence) approaches as it aims at facilitating the understanding of neural networks using visual analytics. Our approach also shares similarities with visualization systems in the sense that it aims at developing a framework for building customized interfaces. In this section, we discuss the existing approaches related to these two topics.

- *Shaoxuan Lai and Wanna Luan are with the School of Computer Science and Engineering, Sun Yat-sen University. E-mail: {laishx3,luanwn}@mail2.sysu.edu.cn.*
- *Jun Tao is with the School of Computer Science and Engineering, Sun Yat-sen University, National Supercomputer Center in Guangzhou, and Southern Marine Science and Engineering Guangdong Laboratory (Zhuhai). E-mail: taoj23@mail.sysu.edu.cn. He is the corresponding author.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

2.1 Vis4AI

The existing Vis4AI approaches are summarized by several surveys. Choo et al. [6] classified the approaches based on their goals, such as education, debugging, and understanding. Rosa et al. [17] categorized the relevant works from the perspective of the types of explanations. Hohman et al. [12] and Alicioglu et al. [1] summarized the works from similar perspectives, including usage, approaches, and users.

In this section, we summarize the work from two perspectives: the data analyzed and the view (i.e., visual representation) provided. Studying the data may inform us what are the data of interest, and studying the view may inform us how the data should be presented. As these two factors may vary across neural network architectures. We will discuss them by architectures, including the recurrent neural network (RNN), the convolutional neural network (CNN), the multilayer perceptron (MLP), the graph neural network (GNN), and the Transformer.

For **data in RNN**, the hidden state is often shown. RNNVis [26], LSTMVis [38], and Seq2SeqVis [37] visualize the hidden state of RNN or its variant long short-term memory network (LSTM). Seq2SeqVis [37] further visualizes the attention. Karpathy et al. [16] visually analyzed the representations, predictions, and error types of LSTM. Shen et al. [34] designed a visual analytics system that measured how feature selections affected the output distribution.

For **data in CNN**, the activation maximization [9, 35, 49] is commonly used to represent the learned features. Wang et al. [46] presented CNNExplainer which visualizes the output of neurons and the convolution operation details. Das et al. [8] designed Bluff using adversarial examples to help show the feature learned by a neuron. DECE [5] and AdViCE [10] use the change from original samples to counterfactual explanations to construct the interface. Li et al. [21] ranked the neurons by their vulnerability levels and identifies image features that highly stimulate a user-selected neuron. GANViz [44] visualizes the samples and activation of different inputs of CNN in the generative adversarial network (GAN).

For **data in Transformer**, attention is usually visualized. Vig et al. [43] visualized attention in Transformer at multiple scales. They introduced a high-level model view for visualizing all layers and attention heads and a low-level neuron view for showing how individual neurons interact to produce attention. Hoover et al. [13] designed exBERT, which used a heatmap and links to visualize the multi-head self-attention of input words. Wang et al. [45] designed DORIIO that tightly integrates an overview of the roles of different attention heads and multiple detailed views for comparing attention weights.

For **data in GNN**, node and topology attract more attention than activations. GNNLens [15] shows the node information and topological structure of the whole graph. Users can also zoom in and focus on the topological structure of a specific node. VisGNN [27] decomposes a visualization into its data and visual components, and then jointly models each of them as a large graph to obtain embeddings of the developers, attributes, and visual configurations. TensorFlow Playground [36] visualizes the activation of grid data to serve as the decision boundary neurons and the connection between layers of **MLP**.

From the perspective of **views**, for hidden states, it is common to perform dimension reduction using t-SNE [41] and then visualize the 2D embedding using a scatter plot, e.g., [29, 37]. A distinct example is that LSTMVis [38] uses parallel coordinates to visualize the hidden state of temporal data. For text data, word clouds [26] and a view of sentence lists [37, 38] are used. Some systems (e.g., GNNLens) include specially designed views for representing graph data. Many systems use line charts to visualize the training metrics of neural networks, e.g., [2, 28, 39]. Harley et al. [11] and Tzeng et al. [40] used DAG to represent the structure of a CNN. CNNVis [23] and Tensorflow Playground [36] use links to represent the connection data. Cao et al. [22] proposed the datapath to represent the important connection between neurons. ReVACNN [7] visualizes the underlying process of a convolutional neural network by monitoring the filter-level 2D embedding view. Chae et al. [4] visualized the classification results of samples with stripes to reduce visual clutter.

As interactive systems, views on the interface are often interrelated through **interaction**. Some systems allow the training process to be adjusted through the interface [36], while some allow data or model modifications through the interface and receive feedback. For example, Seq2SeqVis [37] allows for modifying attention and observing prediction results based on the updated attention, and Tensorflow Playground [36] allows modifications to the connection.

Comparison with existing works. The design of our abstract model and toolkit is inspired by the existing works. We extract common patterns from these works to develop our approach, such as the types of data, the visual representations, and the interactions. We also learn that specialized designs (e.g., the river-based visualization [22]) are often needed, and therefore support customization in many of our modules.

Our work differs from existing approaches in the sense that it aims at rapid prototyping for diverse kinds of network architectures and analysis purposes, instead of building a sophisticated interface for a specific network. As a result, we design a model and an associated visual representation for the data and workflow of neural networks. The model validates and guides the development of our toolkit. It guides the design and implementation of interactive interfaces for non-visualization users. Finally, we provide a network-agnostic toolkit for prototyping interfaces. The model and the toolkit are not available in existing approaches.

2.2 Visualization System

As for the implementation of visualization systems, researchers have developed various visual analytics interfaces for exploring, analyzing, and communicating data. The web application is convenient and has become the most popular platform. Many web-based visualization techniques have been proposed. D3 [3] is a popular JavaScript library for creating interactive visualizations on the web. Vega [33], enables developers to declaratively specify visualizations using a JSON-based grammar. Vega-Lite [32] is a simplified version of Vega, designed to make it easier for developers to create visualizations without sacrificing flexibility. ggplot2 [47] also provides a declarative grammar for specifying visual encoding to create visualizations. Li et al. [18] developed P4, a declarative visualization toolkit for building GPU-accelerated visualization systems. The later version support parallel progressive visualization for P5 [19], and enables using machine learning methods to help process data in P6 [20].

Several Python libraries embed a JavaScript library to generate visualizations. They enable creating visualizations in Python without writing JavaScript. For example, Altair [42] uses Vega-Lite to generate visualizations. Matplotlib [14] is also a popular library for creating visualizations in Python.

Comparison with existing works. The overall design of our toolkit is similar to existing frameworks, but it also differs from them in the emphasis on tensor data and neural networks. Our toolkit adopts a client-server architecture. This separates the framework implementation in Python from the visualization development in JavaScript. This allows the framework to be easily integrated into learning code and still benefits from the rich suite of visualization tools in JavaScript.

3 Design of the Model

We propose a model that abstracts the data, transformations of data, and interactions in visual analytics systems for understanding neural networks. This abstraction model is used to guide the design of our programming toolkit. Our model considers relevant data as tensors with multiple named dimensions and abstracts the interaction process based on the dimensions of tensors. In this section, we first introduce a foundational model framework, which identifies the key components to be studied and guides the design of the exact model. We then review a collection of existing works to extract the ingredients to instantiate the framework and complete the model design. Finally, we provide concrete examples by representing several existing visual analytics interfaces using our model.

3.1 Foundational Model Framework

In the analysis of the foundational model framework, we identify the key components in the design space and the connections among these

Table 1: The types of data, transformations, and responses supported by existing approaches. For data, "T" and "NT" denote tensor and non-tensor data, respectively. For transformations, "F", "A", and "O" denote filtering, aggregation, and other transformations, respectively. For responses, "C", "P", "H", and "A" denote content modification, parameter modification, highlighting, and animation, respectively.

Approach	Data		Transformation			Response			
	T	NT	F	A	O	C	P	H	A
RNNVis [26]	✓		✓	✓		✓	✓	✓	
LSTMVIS [38]	✓	✓	✓		✓	✓		✓	
Seq2SeqVis [37]	✓	✓	✓		✓	✓		✓	
ProtoViewer [50]	✓		✓			✓	✓	✓	
CNNVis [23]	✓		✓	✓		✓	✓	✓	
CNNExplainer [46]	✓		✓			✓	✓	✓	✓
Bluff [8]	✓		✓	✓		✓	✓	✓	
AeVis [22]	✓		✓	✓		✓	✓	✓	
DECE [5]	✓		✓			✓		✓	
AdViCE [10]	✓		✓	✓		✓		✓	
VAC-CNN [48]	✓		✓	✓		✓		✓	
EXBERT [13]	✓	✓	✓	✓	✓	✓	✓	✓	
DODRIO [45]	✓		✓			✓	✓	✓	
TF Playground [36]	✓		✓			✓			✓
GANViz [44]	✓		✓	✓		✓	✓	✓	
GNNLens [15]	✓	✓	✓	✓		✓		✓	

components to be studied. Specifically, we considers the following components: *data*, *transformation*, *views*,and *interaction*.

Data and transformation. Various kinds of data are involved in neural networks, including input samples, outputs, intermediate data, training profiles, and network parameters. The data may be transformed to produce data for further analysis. Transformation should include standard ones (e.g., filtering) and application-specific ones (e.g., decomposing hidden states). Our model will focus on the types of data and transformation instead of the meaning of data or how the data is obtained.

Views and data. Views are basic ingredients to form visual analytics systems. They determine how the data are visually encoded (e.g., mapping attributes to positions or colors). Our model will focus on how the data are collected, organized, and bound to the views. Specifically, we would like to study how to specify data in a neural network for displaying, how to specify data for highlighting, and how to update the views with online data. Note that the online data may be generated on-the-fly by the network or through interactions.

Interaction and others. Interaction is related to all the other three components. Although interactions are generated in views, our model does not use them to modify the visualization directly. Instead, they are used to specify transformations and update the data bound to the views. The change of data is then reflected in the visualization. This design isolates visualization from computation triggered by interactions.

Based on the above analysis, we summarize the foundational framework of our model as follows:

- Data is tightly bound to views.
- Data can be transformed to produce other data.
- Interactions can affect data and transformations, leading to consequent updates of views.

3.2 Literature Review

We review the existing visual analytics interfaces to guide the detailed design of our model. Specifically, we summarize the existing work in the following aspects:

- **Types of data.** For data types, we consider tensor and non-tensor data. Tensors are heavily used in neural networks for representing parameters and samples with regular shapes. Non-tensor data is often involved as input data or samples with irregular shapes, such as graphs and sentences. As tensors are standard and relatively easy to handle, this review mainly analyzes the visualization of non-tensor data in existing works.
- **Types of transformations.** We summarize the transformations involved in existing approaches. Note that the transformation in data preprocessing is independent of the visualization system. Therefore, we do not consider this kind of transitions in our model and treat the preprocessed data as input. Our review focuses on the transformations during exploration, including the typical transformations provided by interactive systems (e.g., brushing for filtering data) and customized procedures as callback functions triggered by interactions.
- **Type of responses.** We summarize how the existing approaches respond to the data transformation and interaction. While we do not consider responses as a component in our model, analyzing the types of responses may guide the design of transformation and interaction.

Our findings are summarized in Tab. 1. In terms of the *data type*, we find that all approaches use tensors, and only four of them involve non-tensor data. Three approaches are language-related models, visualizing words and sentences with varying lengths. The other approach involving non-tensor data is GNNLens [15]. GNNLens targets the graph neural network, which takes graphs as input. We notice that the non-tensor data appear as input data in all these four approaches. This means that the non-tensor data may only be visualized when we need to refer to the original data samples. *For understanding the behavior of networks, it is usually sufficient to analyze tensors.* In addition, we find these approaches may still organize the non-tensor data in a regular shape for visualization. For example, in Fig. 5, LSTMVis [38] displays a fixed number of consecutive words in sentences. *This indicates that we may convert non-tensor data to tensors by truncating and padding.*

For the *transformation type*, we find that filtering and aggregation are the most commonly used. All approaches use filtering and nine out of the sixteen approaches support aggregation. Additionally, three approaches involve other sophisticated transformations. While filtering and aggregation are standard, the other transformations may require specific algorithms. *This indicates our toolkit should provide both built-in transformations and support customized ones through extensions.*

For *responses*, we identify four common types of responses. The first type is modifying the content of a view, such as displaying data selected in interactions. This type of response is used by all approaches. The second is adjusting parameters of a view. For example, an interface may expand a view at the clicked position to show details. This can be seen as modifying the position parameter of the view. The third is highlighting a subset of data in a view, which is adopted by all approaches except TensorFlow Playground [36]. The fourth type is showing animation upon interactions. For example, TensorFlow Playground shows an animation of network parameters and outputs over training steps.

To realize these four types of responses, our model and toolkit should meet the following requirements. First, modifying the content of a view can be achieved by updating the data bound to that view. But we should note that *the binding of data should be dynamic*, meaning that the visualization should be updated automatically after the change of data. Second, highlighting changes the visualization style of a subset of data points. Accordingly, *our toolkit should be able to efficiently describe subsets and visualization styles*. Third, modifying the parameters and animations should be implemented by specific views. Accordingly, *our toolkit should allow users to define their own views*. These two types of responses may require the update of data as well.

3.3 Our Model

Based on the foundational framework and the literature review, we design a tensor-centric model. Non-tensor input data is reorganized, so that they can be processed and visualized as tensors. The data transformation is designed to filter, aggregate, and join (multiple) tensors. The views visually encode different dimensions in tensors, and the interaction specifies transformation based on tensors. Users may extend the transformation, interaction, and views to incorporate non-tensor data. Please refer to the programming toolkit in Sec. 4). In section, we will introduce a visual representation of our model based on tensors. The design of data, transformation, and interaction models will be explained using this representation.

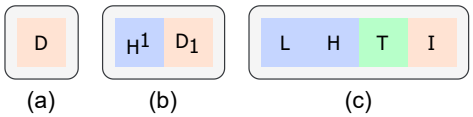

Fig. 1: Visual representation of *tensor data* in our model. Each rounded rectangle represents one tensor bound to a view, and a color box represents one dimension in that tensor. The light red, blue, and green boxes correspond to data-related dimensions, network-related dimensions, and other types of dimensions, respectively.

Data and view. Our model assumes the data to be tensors by default. This means that we consider the data has a regular shape and multiple dimensions. Fig. 1 illustrates three examples of tensor data bound to three views. The data bound to the same view are grouped by a rounded rectangle. In (a), the tensor has a single dimension related to the original data, as indicated by the letter "D" and the red color. In (b), the tensor has two dimensions related to the hidden states ("H") and original data ("D"), respectively. The superscript is used to distinguish multiple dimensions with the same meaning. Here, "H^1" indicates that it is one of the multiple dimensions related to hidden states. The subscript is used to denote a subset extracted along that dimension. Here, "D_1" indicates that the tensor is filtered by the value of original data "D". In (c), the tensor represents the hidden states ("H") of the user input ("I") at the layer ("L") and time step ("T"). This kind of data might be automatically collected by our toolkit during the network training.

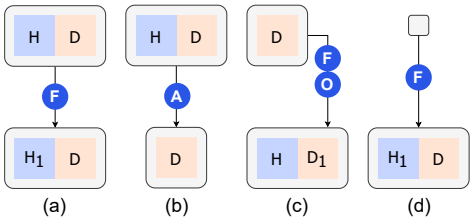

Fig. 2: Visual representation of *transformation* in our model. (a) shows filtering on dimension "H", (b) shows aggregation, and (c) shows filtering followed by other transformations that create another dimension "H". (d) shows a simplified representation of (a) when the upper tensor "H×D" is not visualized.

Transformation. A transformation consumes one tensor and produces another, indicated by a *transformation arrow* between the two tensors. A blue label with a letter on the arrow indicates the transformation type. When multiple transformations are applied, we merge the arrows and use multiple letters to indicate the respective types of transformations. In Fig. 2 (a), an arrow "F" indicates a filtering transformation. Note that we do not state the filtering criterion on the arrow, as the dimension "H_1" in the output already indicates that the data is filtered by the hidden state "H" unambiguously. Similarly, in Fig. 2 (b), we can easily read that the aggregation ("A") is performed along the "H" dimension as it disappears after the transformation. Note that transformation may lead to an extension of dimensions as well. For example, Fig. 2 (c) shows a composite transformation that filters the "D" dimension and brings in another dimension "H". When the input data is not visualized, we may use a small rectangle to represent the input tensor, as shown in Fig. 2 (d).

Interaction. When users interact with a view, an interaction will be created. The interaction carries two pieces of information: a subset from the data bound to that view, and a transformation to be performed. An interaction is represented by an *interaction arrow*. Fig. 3 illustrate three examples of interaction. Fig. 3 (a) shows an interaction that modifies the data in the lower view using a widget (the triangle). Note that the widget and the view may hold data in different spaces. In this case, a mapping must be applied to convert the interacted subset in the widget to a corresponding subset in the view. Fig. 3 (b) shows an interaction involving three views. Users interact with the left view to filter the data in the upper view and visualize the filtered data in the lower view. The subscript "H_1" in the lower view indicates the data are filtered based on the hidden states ("H"). Fig. 3 (c) shows a simpler interaction between two views without transformation.

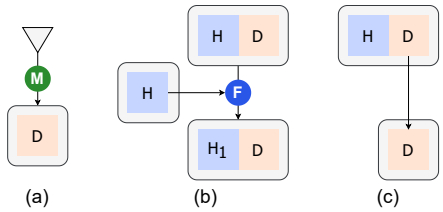

Fig. 3: Visual representation of *interactions*. (a) shows a mapping from a widget to the data in a view. The triangle indicates an interactive widget (e.g., a slider). The green label indicates that the mapping is applied to the interaction data (i.e., subsets). (b) and (c) show two examples where multiple views are involved in the respective interactions.

Compared to existing models for visual analytics, our model centers on the tensor data and the neural network workflow. We also design a concise flowchart representation for the model, that highlights information regarding tensors and transformations. This representation allows us to examine whether our model is able to resemble the existing analytics approaches, and assists non-visualization experts in designing their system. In the future, it may be extended for a visual programming purpose as well (please see Sec. 7).

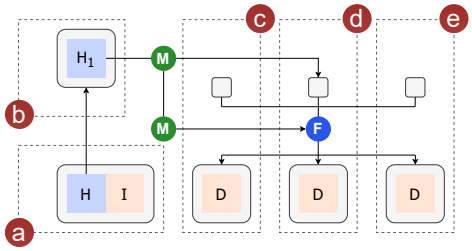

Fig. 4: Using our model to illustrate the design of LSTMVis [38].

3.4 An Illustrative Example: Prototyping LSTMVis

LSTMVis is an interactive visualization system for analyzing a long-short time memory (LSTM) network which processes language inputs. Its analysis aims at discovering the connections between the input languages and the patterns of hidden states produced by the network. In this example, we analyze the design of LSTMVis [38] using our model, as shown in Fig. 4. Please also refer to Fig. 5 for a prototype resembling LSTMVis interface.

Data of concern. We first start with the data required by LSTMVis. LSTMVis concerns the following information: First, the temporal patterns of hidden states. This requires a tensor incorporating the hidden dimensions "H" and the input data "I", as shown in (a). Second, the hidden dimensions with a certain pattern of interest. This requires a tensor representing a subset of dimensions "H_1", as shown in (b). Third, the segment of sentences corresponds to user-selected hidden dimensions. LSTMVis shows the segments of sentences together with two associated attributes. This requires three tensors showing the original data "D", as shown in (c), (d), and (e). This analysis produces the dashed rectangles (i.e., data and corresponding views) in the abstract model. In the next step, we need to connect the rectangles with transformations and interactions.

Transformations and interactions. The exploration using LSTM can be summarized by two tasks: identifying hidden dimensions of interest and observing data related to selected dimensions. For the first task, users observe temporal patterns of hidden states and select

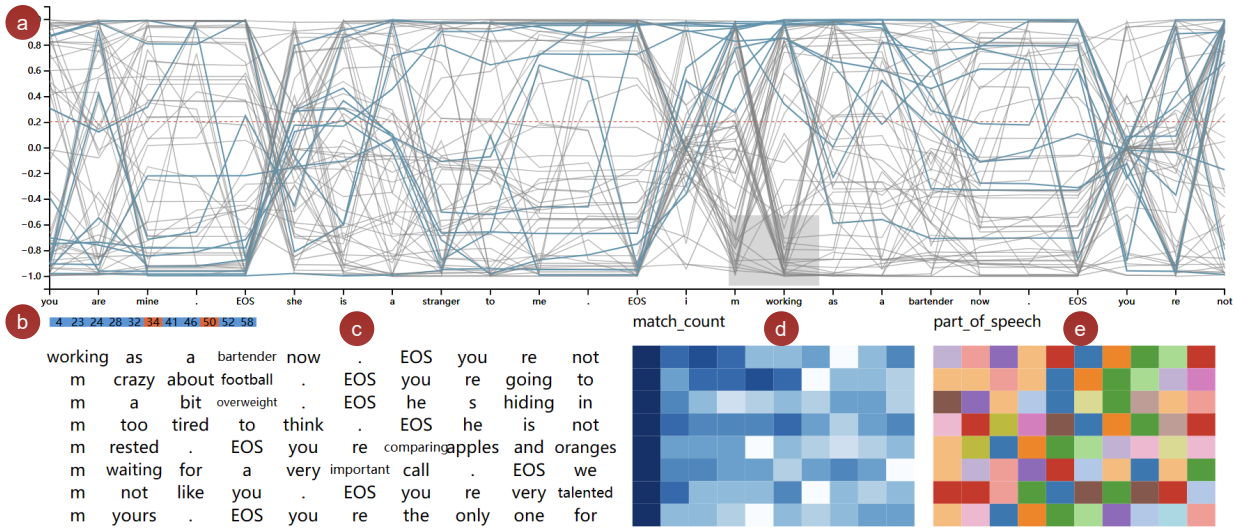

Fig. 5: The prototype resembling LSTMVis built with NNVisBuilder. (a) shows a parallel coordinates plot for embedding. (b) shows a list of dimensions for selection. (c) shows the sentence segments containing the selected words. (d) and (e) show the match count and POS taggers corresponding to the words in (c).

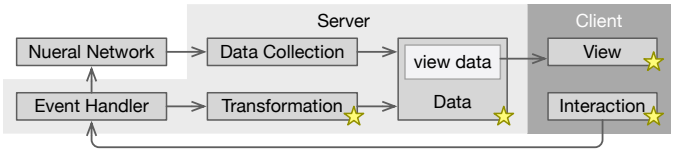

Fig. 6: The architecture of our NNVisBuilder toolkit. The stars highlight modules that should be specified or customized.

hidden dimensions exhibiting desired patterns in (a). The selected ones will be displayed in (b). This leads to the first interaction, represented by the edge from (a) to (b). For the second task, users select hidden dimensions in (b) and observe related sentence segments in (c), (d), and (e). This means that we need to map the selected dimensions to sentence segments (green "M") and filter the segments accordingly (blue "F").

A guide to applying our model. Following this example, users without professional visualization knowledge may use our model to guide the development of their own exploration systems in two steps. First, users may list all the data of concern by views (e.g., the dashed rectangles in Fig. 4). Each view should contain the tensor dimensions (i.e., attributes of data). Second, users may connect the related views using transformations and interactions. The content in views may suggest the appropriate transformation. For example, an edge between different dimensions (e.g., "H_1"→"D") indicates the need for a mapping. Please refer to Sec. 5 for more examples. In the next section, we will explain how our toolkit realizes the abstract model.

4 Programming Toolkit

We design NNVisBuilder as a toolkit to realize the abstract model described in Sec. 3. This section starts with an overview of the toolkit and is followed by the design and usage of each module.

4.1 Overview

NNVisBuilder adopts a client-server model, as shown in Fig. 6, where the client is responsible for visualization and interaction, and the server is responsible for data management.

Client and views. In our current implementation, the client is a web interface, consisting of a suite of views developed using D3 [3]. Users may implement their own views for customization as well. When users interact with the views, the client will create an interaction event and send the event to the server.

Server and data flow. The server collects data from the target neural network (e.g., samples, labels, latent encoding, and network parameters), and organizes the data in the form of tensors. It also handles the interaction events from the client. This may trigger transformations (e.g., filtering) to update the tensors managed by the server or invoke external computations (e.g., tuning the network) to update existing data or produce new data. The server is implemented in Python. It currently supports neural networks developed using PyTorch [30], which is extensively used in the deep learning community.

Development with NNVisBuilder. To build a customized interface with NNVisBuilder, users need to specify: which data to visualize and explore, how are the data visualized, and what are the interactions and transformations, as shown by the starred modules in Fig. 6. In the abstract model (e.g., Fig. 4), the specification of data and views creates the dashed rectangles, and the interactions and transformations specify the arrows. In this section, we will explain the design and usage of NNVisBuilder using the prototype of LSTMVis as an example.

4.2 Data and View

NNVisBuilder can collect and manage data from the target neural network, including embedding (i.e., hidden states produced from samples), connections (i.e., network parameters), and gradients for developers. Developers only need to define an NNVisBuilder object. They can bind the model to this object and specify their data of interest. The NNVisBuilder object will then collect and manage the specified data. It will keep track of the managed data and report the update to views where the data are bound. It can also record the history of managed data for later investigation on demand. Please refer to Fig. 7 (0) for an example of binding the builder object to the network and adding embedding for management in LSTMVis.

Creating data objects. NNVisBuilder provides a unified interface for managing static and dynamic data. The static data are independent to the interaction (e.g., data extracted from the network). Therefore, they can be produced before the client is launched. For example, in Fig. 5, the view (a) of LSTMVis shows the hidden embedding of words, which is available before defining the corresponding view. In this case, users may provide the values to construct the data objects 'word' and 'embedding', as shown in Fig. 7 (a). NNVisBuilder also supports out-of-memory storage through external files, which helps users build interfaces for large-scale models.

For data that are generated during the interaction, NNVisBuilder still allows users to create the data objects and bind to the views before the data are available. For example, in Fig. 5, the view (b) of LSTMVis

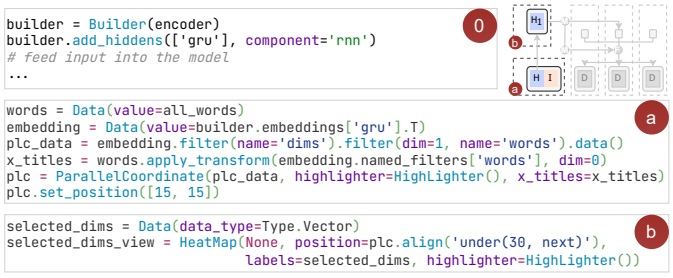

Fig. 7: The code for the data collection and view creation in the prototype of LSTMVis. The thumbnail at the corner highlights the corresponding modules in the abstract model (Fig. 4). The code block in (0) collects data from the target neural network. The code in (a) and (b) creates the respective views in the interface (Fig. 5).

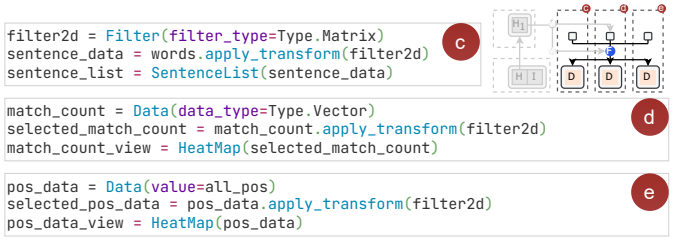

Fig. 8: The code for the transformation definition and view creation in the prototype of LSTMVis. The thumbnail highlights the corresponding modules in Fig. 4. (c) defines the transformation and one view. (d) and (e) defines two additional views using the transformation.

shows the indices of selected dimensions, which is not available when constructing the interface. In this case, users may simply define a data object 'selected_dims' by specifying its data type without providing the values, as shown in Fig. 7 (b). The values of this object will be updated by a transformation triggered by interactions, which will be explained in Sec. 4.3.

Creating Views. A view can be created easily with the corresponding data object provided. For example, in Fig. 7 (a) and (b), the parallel coordinates view to visualize hidden states and the heatmap to show selected dimensions are each created with one line of code.

In NNVisBuilder, the views provide customizable modules to specify their behavior. For example, the highlighter specifies how selected data should be highlighted in a view. Users may use the default highlighting scheme of a view by providing an empty highlighter, as shown in Fig. 7. Alternatively, they may create their own highlighter for each view as well. In addition to the data objects and modules, the views have many other associated attributes. These attributes may be provided in the object construction or specified at a later step. For example, in Fig. 7, the parallel coordinates view 'plc' does not specify a position during construction and a default position will be used. Users may specify a coordinate as the position explicitly later. Users may also align a view with an existing one, Please see the construction of the selected dimensions view ('selected_dim_view') in Fig. 7 (b).

Data transitions to views. A view can bind multiple data objects corresponding to different visual channels. When binding a view with a data object, NNVisBuilder will record this relationship between the view and the data. Once the data is produced or updated, the server will convert the tensor data into DataFrame [24] objects and send it to the client in JSON format. To tackle large-scale data, users may apply filters to the data object to reduce its size. Random sampling or other sophisticated sampling approaches may be used as filters. Note that we consider the data reduction as a user-specified option and will not apply any filtering or sampling automatically.

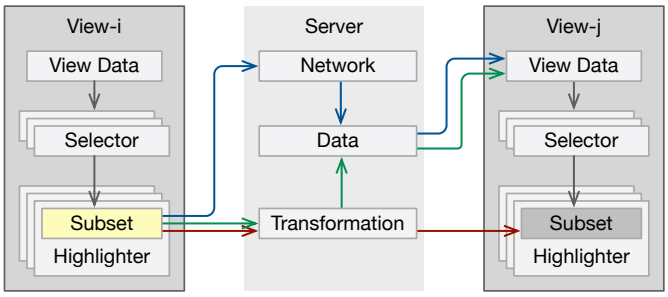

Fig. 9: Interaction mechanism of NNVisBuilder. The red, blue, and green arrows illustrate three common pathways in the interaction workflow of NNVisBuilder, respectively.

4.3 Transformation

Similar to the data object, the transformation object can be defined statically with available data or dynamically without actual data. For a dynamic transformation object, the transformation will be performed when the required data is provided in interaction events. The same transformation can be applied to multiple data. We will use LSTMVis as an example to explain these two properties (i.e., applying a single dynamic transition to multiple data).

Usage example. In Fig. 5 (c), (d), and (e), the visualized data are produced by filtering three different tensors based on the user-selected patterns. For example, the sentences ('sentence_data') are filtered from all input words ('words') and visualized in the view ('sentence_list'). In this case, both the data and transformation related to the sentence view should be generated dynamically during interaction. With NNVisBuilder, users may simply define the dynamic transformation object 'filter2d' by specifying its type without any data. Users may further apply this transformation to produce the dynamic data object 'sentence_list', and use this data object to create the view. Once the transformation is performed during interaction, the data and the visualization will be automatically updated. The same filtering transformation can be applied to the match count and Part-Of-Speech (POS) tagger data of sentences to extract information associated to the selected dimensions.

Transformations supported. NNVisBuilder provides two basic transformations (i.e., filtering and aggregation), implements several complex transformations, and supports customized transformations. For filtering and aggregation, users need to specify the tensor and dimension to apply the transformation. Aggregation requires a further type of operation to specify how the data is aggregated, such as summation, averaging, and computing maximum. The complex transformations involved in neural network visualization are often application-specific. Therefore, we only implement some algorithms as examples, such as activation maximization.

4.4 Interaction

The interaction mechanism is shown in Fig. 9. An interaction event is invoked when users interact with elements in a view. This process starts from the selector of the view, which is responsible to identify the subset of interacted data points. This interacted subset will be passed to a highlighter, which updates their visualization style. The selector and highlighter define how a view reacts to interaction with itself.

Typical data update pathways. The interaction involving multiple views may follow three typical pathways, as illustrated by the red, blue, and green arrows in Fig. 9, respectively. Following the *red pathway*, the interacted subset is transformed and highlighted in another view (e.g., brushing and linking). Note that the transformation is optional. It is usually required when the two views visualize data in different spaces and can be omitted when the data are in the same space. Following the *green pathway*, the interacted subset is used to update the data visualized in another view. For example, in LSTMVis, the selected dimensions are used to identify sentence segments. Following the *blue pathway*, the interacted subset is processed by the network and updates

```
def f(value):
    selected_dims.update(value)
    selected_dims_view.highlighter.update(list(range(len(value))))
plc.highlighter.add_mapping(f)
```
g

```
def f(value):
    r = []
    # compute match_count for each word and save in the variable r
    ...
    match_count.update(r)
    s = match_count.argsort(reverse=True)[:8]
    filter2d.from_1d(s, 10, words.size()[0])
selected_dims_view.highlighter.add_mapping(f)
```
h

Fig. 10: The code for the interaction in LSTMVis prototype. The thumbnail highlights the corresponding interaction arrows. (g) defines the interaction between the parallel coordinates view and the selected dimensions view. (h) defines the interaction between the selected dimensions view and the three views related to the selected sentence segments.

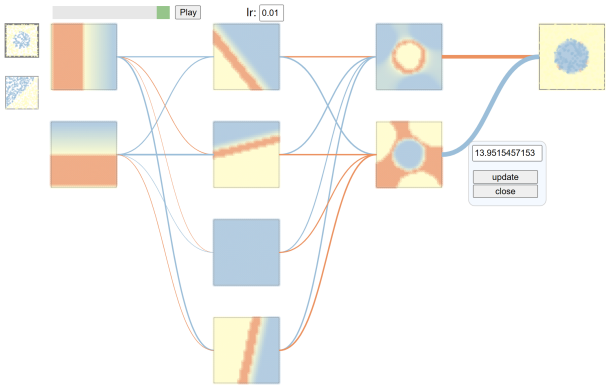

Fig. 11: The prototype of TensorFlow Playground [36]. The sample, the decision boundaries, and the inter-layer weights are shown.

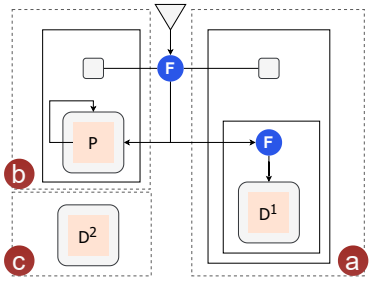

Fig. 12: Using our model to illustrate the design of TensorFlow Playground. The rectangles containing views with solid boundaries denotes repeating pattern.

the data in another view. Users may define their own pathway as a mapping of the highlighter.

Selectors and highlighters. A *selector* defines which tensor dimensions are used to select the items. For example, for a heatmap visualizing a second-order tensor, the selector may select rows of data points along the first dimension of the tensor or columns of points along the second dimension. NNVisBuilder provides various kinds of selector templates for different types of views.

A *highlighter* defines the visualization style of the interacted subset. By specifying the highlighter style, users may change the size or color of the selected subset. A highlighter can be shared by multiple views. In this case, the selected subset is automatically synchronized across views without invoking additional updates of data.

NNVisBuilder provides an additional *multi-selector* and *multi-highlighter* mechanism, which allows composite selection and highlighting rules to be defined using preset templates. For example, the scatter plot provides templates for selecting points to the left and above the brushed region. Users can combine these two selectors to create a multi-selector that selects the intersection or union of these points. Similarly, the multi-highlighter can customize the visualization styles of multiple selected subsets. For example, a multi-highlighter can assign two different colors to two consecutively selected subsets and the third color to their intersection. This may be handy to support comparative visualization in a superposition manner. Besides, NNVisBuilder supports view duplication, which may be useful for creating small multiples and facilitate comparisons in a juxtaposition manner.

Usage example. Fig. 10 shows the code to add interactions to our LSTMVis prototype. The code in Fig. 10 (g) defines the interaction between the parallel coordinates view ('plc') and the selected dimensions ('selected_dims_view'). The mapping function 'f' updates the data in the selected dimensions view. By associating 'f' with the highlighter of the 'plc' view, the function 'f' will be called when data are selected in the 'plc' view. This realizes the red path in Fig. 9. The code in Fig. 10 (h) defines the interaction between the selected dimensions view and the three views showing attributes of corresponding sentence segments. Similarly, we associate the highlighter of the selected dimensions view with a mapping function 'f'. This mapping function computes the corresponding match count values and modifies the filter ('filter2d') to extract sentence segments containing the identified word. This realizes the green path in Fig. 9.

5 Evaluation

In this section, we present two additional example applications to demonstrate the capabilities and usefulness of NNVisBuilder for developing prototypes of interfaces for different neural network architectures.

5.1 TensorFlow Playground

Analyzing the interface. We demonstrate how to use NNVisBuilder to build a prototype of TensorFlow Playground, which supports animation and interactive updates of the network. TensorFlow Playground visualizes the parameters and output of an MLP model over training steps. The interface created by NNVisBuilder is shown in Fig. 11. The two rectangles in the leftmost column represent two different datasets. Users may switch datasets by clicking the corresponding rectangle. The three columns in the middle display the decision boundaries at three layers, respectively. Each rectangle corresponds to one dimension of the layer's output. The rightmost view shows the final classification results. This interface allows users to modify the learning rate and the weights of the network. It also allows users to specify a training step for visualization or animate through the entire training process.

Model. TensorFlow Playground visualizes a multi-layer perceptron (MLP) network, as shown in Fig. 11. It concerns three types of information: (a) the inter-layer connections, (b) the decision boundaries of each dimension at each layer, and (c) the labels, as shown in Fig. 12. Note that we use a rectangle with a solid boundary to denote small multiples without repeating the view. A slider is used to filter the decision boundaries, and an input box is used to update the learning rate of the network, leading to the transformations in (b).

Building the interface. NNVisBuilder provides templates for generating grid data and encapsulates the process of computing decision boundaries from the grid data. Note that NNVisBuilder will not automatically record the history of every parameter, and users need to explicitly invoke the data collection in the training process. In this example, we collect the connections and decision boundaries. The data collection module will automatically organize the data, so that the time step can be used to access corresponding data.

Fig. 13 shows our code to create views and define interactions. The code in (a) iterates through all layers and dimensions to create the respective view of decision boundaries. Each view is associated with a transformation to filter data along the time dimension (i.e., training step). The last five lines of code create views for the final result, x and y signals, and two datasets, respectively. The data in these views do not change over training steps. The code in (b) adds link views to connect neighboring layers. Each link view is associated with an 'onclick'

```
filter = Filter(dim=0)
for i in range(len(layers)):
    layer = layers[i]
    embeddings.append(Data(value=builder.embeddings_g[layer]))
    # reshape to generate time dimension
    e_data = embeddings[-1].reshape(...)
    for j in range(out_features[i]):
        # create one view for each dimension in each layer
        view_data = e_data.apply_transform(filter).filter(dim=1, value=j).data()
        view = ScatterPlot(grid, color_labels=view_data)
view_final = ScatterPlot(input_x, position=view.align(), color_labels=input_y)
view_x = ScatterPlot(grid, color_labels=grid[:, 0])
view_y = ScatterPlot(grid, color_labels=grid[:, 1])
view_dataset1 = ScatterPlot(input_x1, color_labels=input_y1)
view_dataset2 = ScatterPlot(input_x2, color_labels=input_y2)
```

(a)

```
for i in range(len(layers)):
    connections.append(Data(value=builder.connections[layers[i]]))
    link_labels = connections[-1].apply_transform(filter).reshape(-1)
    # Define the endpoint position of links
    node_positions = Data(value=...)
    link = LinkView(node_positions=node_positions, labels=link_labels,
                    colors=link_labels.sign(), width='labels')
    link.onclick(get_modify_weight(i, link_labels))
```

(b)

```
slider = Slider(range=time_steps)
def f(value):
    filter.update(value)
slider.onclick(f)
```

(c)

```
def f(value):
    builder.reset_embedding()
    # train model using dataset 1
    ...
    for i in range(len(layers)):
        layer = layers[i]
        embeddings[i].update(builder.embeddings_g[layer])
        connections[i].update(builder.connections[layer])
    input_x.update(x_train1)
    input_y.update(y_train1)
view_dataset1.onclick(f)
```

(d)

Fig. 13: The code used to build the prototype of TensorFlow Playground. (a) defines the views for different layers and inputs. (b) defines the link views between layers. (c) defines a slider to select a training step. (d) defines the behavior of selecting a dataset.

function that specifies how the network updates the data (i.e., the blue pathway in Fig. 9). The code in (c) and (d) specifies the interactions corresponding to the slider and the dataset. Note that NNVisBuilder may change parameters or training samples of the network, but it does not support modification of the network architecture (e.g., changing the number of layers). Users need to change the network model and re-run NNVisBuilder.

5.2 Customized CNN Visualization

Previous examples analyze existing systems and build prototypes of them. In this example, we customize a visual analytics interface for a convolutional neural network (CNN), as shown in Fig. 14.

Model. The model of the analytics interface for CNN is shown in Fig. 15. In terms of data, we are interested in: (a) the picture class, (b) the convolution output, (c) the convolution kernels, (d) the activations, (e) original images, and (f) final outputs. For the interaction, we would like to: First, select a picture class and show data related to that class. This leads to the two filtering transformations from (a). Second, select a kernel and show the related convolution results. This leads to the arrow from (c) to (b). Third, select samples from latent space and see the corresponding images.

Building the interface. Then we realize the model using NNVisBuilder. The source code is shown in Fig. 16. The code in (a) defines the slide and corresponding interaction. This interaction updates a filter, which is bound to view (b) and (e). Therefore, when the slider is changed, the content in (b) and (e) will be updated. The code in (b) defines a tooltip, which is shown when the kernel in (c) is selected. The first three lines in (c) define an event handler that brings in the tooltip (b). As there are multiple channels in the kernel, a maximum aggregation is applied to the image data to ensure that the strong activation signals are not omitted. For the embedding views in (d), t-SNE is used to reduce the dimension of the latent spaces to 2D, so that they can be visualized using scatter plots. Note that, to draw images, our current implementation of NNVisBuilder will store temporary images

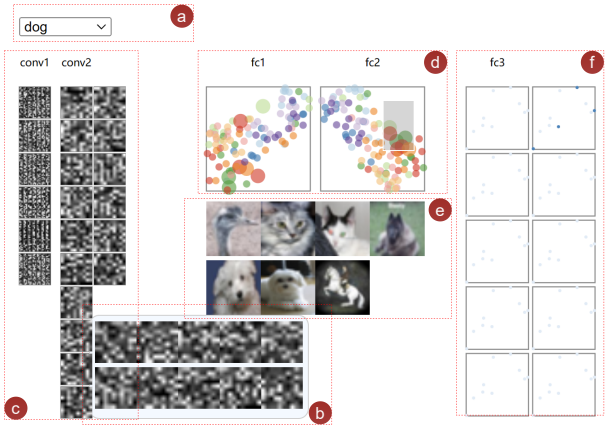

Fig. 14: A customized interface for understanding CNN using image data. (a) shows a widget for selecting image class. (b) shows the convolution result of a selected sample using a selected kernel. (c) shows the convolution kernels. (d) shows the 2D embedding of activations. (e) shows the image samples. (f) shows the output at the last layer.

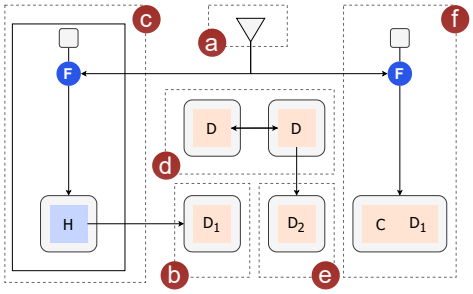

Fig. 15: Using our model to analyze the structure of the visual analytics interface for CNN.

and send the file paths to the client for display. This may damage the performance of large scale datasets. In addition to data reduction through filtering and sampling, we will incorporate modules to display the images directly from data in future extensions.

6 Target Users and Usage Guide

In this section, we will explain the goal of NNVisBuilder and identify the potential users that may benefit from the tool. We will then discuss how the potential users may use the tool.

Design goal. Our NNVisBuilder targets the usage scenario where rapid prototyping of visual analytics interface for understanding neural networks is desired. It incorporates the common types of data, transformations, and interactions that are used in existing visual analytics systems or neural networks. It facilitates easy data management that is tightly bound to the data flow of neural networks. It reduces the effort for writing data and interaction management code, and allows developers to focus on the design of the interface.

But we should also note that NNVisBuilder is not designed to replace the specialized visual analytics approaches. In particular, NNVisBuilder may not reduce the effort for developing interfaces with complicated visual design, such as seamlessly organizing many special glyphs. In this case, it can still be costly for users to map the tensor data to the complex visual representation, and NNVisBuilder may be less helpful.

Target users. NNVisBuilder targets users with the need for rapid prototyping, who are less concerned about fine-level visual representation. These users may include:

First, *practitioners in the learning community* are the key target users. NNVisBuilder allows them to investigate the learned parameters, outputs, and intermediate results in desired manners. As the network may be studied from different perspectives, NNVisBuilder is helpful

```
class_filter = Filter(dim=0, value=0)
select = Select(options=classes)
def f(value):
    class_filter.update(value)
select.onclick(f)
```
a

```
t_data = Data(value=list(range(sub_size)))
t_prefix = Data(data_type=Type.Scalar)
toolTip = Tooltip(t_data, prefix=t_prefix)
```
b

```
def handler(value, position):
    t_prefix.update('%s-%d' % (value, class_filter.value_()))
    toolTip.set_position(position)

layers = ['conv1', 'conv2', 'fc1', 'fc2', 'fc3']
for i in range(2):
    layer = layers[i]
    embedding = builder.embeddings[layer]
    for j in range(builder.name2module[layer].out_channels):
        for c in range(num_class):
            pics = Data(embedding[c*10:(c+1)*10, j, :, :])
            # save pictures of all conv output
            for k in range(sub_size):
                Data(pics[k]).save_img(k, prefix='%s-%d-%d' % (layer, j, c))
            # save pictures of aggregation with different prefix
            pics.aggregation(op='max').data()\
                .save_img(c, prefix='%s-%d' % (layer, j))
        pic_view = Picture(class_filter, prefix='%s-%d' % (layer, j))
        pic_view.onclick(handler)
```
c

```
hl = HighLighter(style='circle_size')
for i in range(2, 4):
    embedding = builder.embeddings[layers[i]]
    data = Data(TSNE().fit_transform(embedding))
    sp = ScatterPlot(data, color_labels=labels, highlighter=hl, title=layers[i])
```
d

```
g_data = Data(data_type=Type.Vector)
gallery = Gallery(g_data)
def f(value):
    g_data.update(value)
hl.add_mapping(f)
```
e

```
embedding = builder.embeddings[layers[-1]].reshape(num_class, sub_size, -1)
img_pos = Data(TSNE().fit_transform(images.cpu().view(...)))
for i in range(embedding.shape[2]):
    pos = img_pos.apply_transform(class_filter)
    labels = Data(embedding[:, :, i]).apply_transform(class_filter)
    sp = ScatterPlot(pos, color_labels=labels)
```
f

Fig. 16: The code for building the customized interface for CNN. Each code block defines the respective view in Fig. 14.

for them to agilely switch the "viewing angle" to examine the networks.

Second, *tutors in the learning community* may benefit from NNVisBuilder as well. NNVisBuilder allows them to illustrate the behavior of networks of different architectures. More importantly, when students raise questions regarding specific components of a network, the tutor may edit a few lines of code to provide new perspectives.

Third, NNVisBuilder may facilitate the *collaboration between researchers in learning and visualization communities*. Although specialized visual analytics approaches are more effective on their respective target problems, NNVisBuilder may still be helpful to examine the research goals and tasks in the initial stage. Additionally, we should note that the collaboration with visualization experts may only be available to a small amount of researchers in learning. Therefore, NNVisBuilder may still hold its unique value.

Guide for practitioners in the learning community. Non-visualization experts may use NNVisBuilder without modifying the visual representations. As described in Sec. 3.4, they may start by listing all the data of concern and organizing them by views. Then, they may identify the transformations and interactions required to connect the views. These steps will produce an abstract model. They may follow the examples in this paper to identify the transformation and interaction available to realize the model. We also provide a step-by-step example guide for them to learn the usage. This guide is provided in both textual and video formats. In addition, a complete description of the API is available on our GitHub repository as well. Please refer to the supplemental material for details.

Guide to extending NNVisBuilder. Although NNVisBuilder mainly targets non-visualization users, we welcome visualization developers to contribute to possible extensions. Both the visual representation and the interaction are customizable. Developers may redefine the selector and highlighter to customize the interaction behavior. They may also specify interaction mappings and callback functions, as shown in previous examples. Here, we provide an example to explain how to create a customized view using the scatter plot as an example, as shown in Fig. 17. Developers need to overload the base class 'View' and simply implement two functions that bridge the Python server and the D3 [3] interface. The function 'generate_vis_data' converts the tensor data to JSON format, and the function 'core' specifies how the JSON data are visualized. Developers may refer to the supplemental material for the API and other details.

```
class MyView(View):
    def __init__(self, data, position, size, ...):
        super(MyView, self).__init__(data, position, size, ...)
        # ...
```
a

```
    def generate_vis_data(self):
        vis_data = pd.DataFrame()
        vis_data['cx'] = self.data[:, 0]
        vis_data['cy'] = self.data[:, 1]
        vis_data['idx'] = list(range(len(self.data)))
        # ...
        return vis_data.to_json(orient='record')
```
b

```
    def core(self):
        super(MyView, self).core()
        self.draw(f"""
            g{self.idx}.selectAll('circle')
                .data(vis_data).enter().append('circle')
                .attr('cx', d => d.cx)
                .attr('cy', d => d.cy)
                .attr('idx', d => d.idx)
                // ...
        """)
```
c

Fig. 17: The code for defining a customized view, using the scatter plot as an example. (a) overloads the base class. (b) converts the tensor data to visualization data. (c) visually encodes the data using attributes in the D3 view [3].

7 Limitations and Future Work

We discuss future work with the limitations of NNVisBuilder as follows:

Support for modifying network structure. Currently, NNVisBuilder only enables modifications to networks' parameters and hyperparameters. If the developer wants to design an interface to compare the results of different network structures, they can only generate separate interfaces for each network and compare them individually. Modifying the network structure requires deep customization for various types of networks, and it is an important part of our future work.

Improving performance. NNVisBuilder uses a client-server architecture, and visualization is handled by D3. When users apply the prototype to large-scale networks and datasets or achieve animations by modifying filters, the real-time performance of the interface cannot be guaranteed. We plan to migrate visualization to the GPU in the future to improve performance.

Visual programming. The visual representation of our model may be used to support visual programming and further reduce the coding effort. The current representation describes the views and their associated data, and specifies the interactions among views. This allows most of the interface specifications to be given in the model representation, such as the type of view, the exact tensor data, and the type of basic transformations. In this case, users may only need to write a minimum of code to specify the complex transformations or overload existing base classes.

8 Conclusion

We proposed NNVisBuilder, a toolkit for rapidly prototyping visual interfaces for different types of neural networks. The toolkit is designed with three modules: data, view, and interaction, and users can design visualizations based on these modules. We also proposed a model for summarizing and comparing neural network analytic interfaces. The model can also guide developers in their coding. To demonstrate the applicability of our model, we provided three example applications where we developed and analyzed prototypes for different neural network analytic interfaces. Furthermore, we discuss future work with the limitations of NNVisBuilder. The source code and some example cases can be found at https://github.com/sysuvis/NVB.

Acknowledgments

This work is supported by National Key R&D Program of China through grant 2021YFB0300103, and the National Natural Science Foundation of China through grants 61902446, 62172456, and 91937302.

Supplemental Material

Our paper is associated with the following supplemental materials:

- **tutorial.pdf**: a step-by-step tutorial for building an interface with NNVisBuilder.
- **manual.pdf**: an introduction to the modules, principles, and API references of NNVisBuilder.
- **video.mp4**: a video demonstrating the example in the tutorial and the construction of LSTMVis prototype.

References

[1] G. Alicioglu and B. Sun. A survey of visual analytics for explainable artificial intelligence methods. *Computers & Graphics*, 102:502–520, 2022. doi: 10.1016/j.cag.2021.09.002 1, 2

[2] S. Amershi, J. Fogarty, A. Kapoor, and D. Tan. Examining multiple potential models in End-User interactive concept learning. In *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*, p. 1357–1360. ACM, New York, 2010. doi: 10.1145/1753326.1753531 2

[3] M. Bostock, V. Ogievetsky, and J. Heer. D³ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185 1, 2, 5, 9

[4] J. Chae, S. Gao, A. Ramanathan, C. A. Steed, and G. Tourassi. Visualization for Classification in Deep Neural Networks. In *Workshop on Visual Analytics for Deep Learning*. IEEE, October 2017. 2

[5] F. Cheng, Y. Ming, and H. Qu. DECE: Decision explorer with counterfactual explanations for machine learning models. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1438–1447, 2021. doi: 10.1109/TVCG.2020.3030342 2, 3

[6] J. Choo and S. Liu. Visual analytics for explainable deep learning. *IEEE Computer Graphics and Applications*, 38(4):84–92, 2018. doi: 10.1109/MCG.2018.042731661 2

[7] S. Chung, S. Suh, C. Park, K. Kang, J. Choo, and B. C. Kwon. ReVACNN: Real-Time visual analytics for convolutional neural network. In *Proceedings of ACM SIGKDD Workshop on Interactive Data Exploration and Analytics*, vol. 14, 2016. 2

[8] N. Das, H. Park, Z. J. Wang, F. Hohman, R. Firstman, E. Rogers, and D. H. P. Chau. Bluff: Interactively deciphering adversarial attacks on deep neural networks. In *Proceedings of IEEE Visualization Conference*, pp. 271–275. IEEE, Salt Lake City, 2020. doi: 10.1109/VIS47514.2020.00061 2, 3

[9] D. Erhan, Y. Bengio, A. Courville, and P. Vincent. Visualizing higher-layer features of a deep network. *Technical Report, Univeristé de Montréal*, 1341(3):1, 2009. 1, 2

[10] O. Gomez, S. Holter, J. Yuan, and E. Bertini. AdViCE: Aggregated visual counterfactual explanations for machine learning model validation. In *Proceedings of IEEE Visualization Conference*, pp. 31–35. IEEE, New Orleans, 2021. doi: 10.1109/VIS49827.2021.9623271 2, 3

[11] A. W. Harley. An interactive Node-Link visualization of convolutional neural networks. In *Proceedings of Advances in Visual Computing*, pp. 867–877. Springer, Cham, 2015. doi: 10.1007/978-3-319-27857-5_77 2

[12] F. Hohman, M. Kahng, R. Pienta, and D. H. Chau. Visual analytics in deep learning: An interrogative survey for the next frontiers. *IEEE Transactions on Visualization and Computer Graphics*, 25(8):2674–2693, 2019. doi: 10.1109/TVCG.2018.2843369 1, 2

[13] B. Hoover, H. Strobelt, and S. Gehrmann. exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformer Models. In *Proceedings of Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 187–196. Association for Computational Linguistics, Online, July 2020. doi: 10.18653/v1/2020.acl-demos.22 2, 3

[14] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(03):90–95, 2007. doi: 10.1109/MCSE.2007.55 2

[15] Z. Jin, Y. Wang, Q. Wang, Y. Ming, T. Ma, and H. Qu. GNNLens: A visual analytics approach for prediction error diagnosis of graph neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 29(6):3024–3038, 2023. doi: 10.1109/TVCG.2022.3148107 2, 3

[16] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and Understanding Recurrent Networks. *CoRR*, abs/1506.02078, 2015. doi: 10.48550/arXiv.1506.02078 2

[17] B. La Rosa, G. Blasilli, R. Bourqui, D. Auber, G. Santucci, R. Capobianco, E. Bertini, R. Giot, and M. Angelini. State of the art of visual analytics for explainable deep learning. *Computer Graphics Forum*, 42(1):319–355, 2023. doi: 10.1111/cgf.14733 1, 2

[18] J. K. Li and K.-L. Ma. P4: Portable parallel processing pipelines for interactive information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 26(3):1548–1561, 2020. doi: 10.1109/TVCG.2018.2871139 2

[19] J. K. Li and K.-L. Ma. P5: Portable progressive parallel processing pipelines for interactive data analysis and visualization. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1151–1160, 2020. doi: 10.1109/TVCG.2019.2934537 2

[20] J. K. Li and K.-L. Ma. P6: A declarative language for integrating machine learning in visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):380–389, 2021. doi: 10.1109/TVCG.2020.3030453 2

[21] Y. Li, J. Wang, T. Fujiwara, and K.-L. Ma. Visual analytics of neuron vulnerability to adversarial attacks on convolutional neural networks. *ACM Transactions on Interactive Intelligent Systems*, 2023. doi: 10.1145/3587470 2

[22] M. Liu, S. Liu, H. Su, K. Cao, and J. Zhu. Analyzing the noise robustness of deep neural networks. In *Proceedings of IEEE Conference on Visual Analytics Science and Technology*, pp. 60–71. IEEE, Berlin, 2018. doi: 10.1109/VAST.2018.8802509 1, 2, 3

[23] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu. Towards better analysis of deep convolutional neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):91–100, 2017. doi: 10.1109/TVCG.2016.2598831 2, 3

[24] W. McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011. 6

[25] P. Migdał, J. Chapman, S. Paul, and R. RV. torchviz: a package for neural network visualization and debugging. `https://github.com/szagoruyko/pytorchviz`, 2019. Accessed on: 2023-03-24. 1

[26] Y. Ming, S. Cao, R. Zhang, Z. Li, Y. Chen, Y. Song, and H. Qu. Understanding hidden memories of recurrent neural networks. In *2017 IEEE Conference on Visual Analytics Science and Technology*, pp. 13–24. IEEE, Phoenix, 2017. doi: 10.1109/VAST.2017.8585721 2, 3

[27] F. Ojo, R. A. Rossi, J. Hoffswell, S. Guo, F. Du, S. Kim, C. Xiao, and E. Koh. VisGNN: Personalized visualization recommendation via graph neural networks. In *Proceedings of the ACM Web Conference*, p. 2810–2818. ACM, New York, 2022. doi: 10.1145/3485447.3512001 2

[28] J. G. S. Paiva, W. R. Schwartz, H. Pedrini, and R. Minghim. An approach to supporting incremental visual data classification. *IEEE Transactions on Visualization and Computer Graphics*, 21(1):4–17, 2015. doi: 10.1109/TVCG.2014.2331979 2

[29] C. Park, S. Yang, I. Na, S. Chung, S. Shin, B. C. Kwon, D. Park, and J. Choo. VATUN: Visual Analytics for Testing and Understanding Convolutional Neural Networks. In *Proceedings of EuroVis Short Papers*. The Eurographics Association, 2021. doi: 10.2312/evs.20211047 2

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems*, pp. 8024–8035, 2019. doi: 10.48550/arXiv.1912.01703 5

[31] L. Roeder. Netron. Computer software, n.d. 1

[32] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017. doi: 10.1109/TVCG.2016.2599030 2

[33] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2016. doi: 10.1109/TVCG.2015.2467091 2

[34] Q. Shen, Y. Wu, Y. Jiang, W. Zeng, A. K. H. LAU, A. Vianova, and H. Qu. Visual interpretation of recurrent neural network on multi-dimensional

time-series forecast. In *Proceedings of IEEE Pacific Visualization Symposium*, pp. 61–70. IEEE, Tianjin, 2020. doi: 10.1109/PacificVis48177.2020.2785 2

[35] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In Y. Bengio and Y. LeCun, eds., *Proceedings of International Conference on Learning Representations (Workshop)*, 2014. doi: 10.48550/arXiv.1312.6034 1, 2

[36] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg. TensorFlow Playground. https://playground.tensorflow.org/, 2017. 1, 2, 3, 7

[37] H. Strobelt, S. Gehrmann, M. Behrisch, A. Perer, H. Pfister, and A. M. Rush. Seq2seq-Vis: A visual debugging tool for sequence-to-sequence models. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):353–363, 2019. doi: 10.1109/TVCG.2018.2865044 1, 2, 3

[38] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. LSTMVis: A tool for visual analysis of hidden state dynamics in recurrent neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):667–676, 2018. doi: 10.1109/TVCG.2017.2744158 1, 2, 3, 4

[39] G.-D. Sun, Y.-C. Wu, R.-H. Liang, and S.-X. Liu. A survey of visual analytics techniques and applications: State-of-the-art research and future challenges. *Journal of Computer Science and Technology*, 28:852–867, 2013. doi: 10.1007/s11390-013-1383-8 2

[40] F.-Y. Tzeng and K.-L. Ma. Opening the black box - data driven visualization of neural networks. In *Proceedings of IEEE Visualization*, pp. 383–390. IEEE, Minneapolis, 2005. doi: 10.1109/VISUAL.2005.1532820 2

[41] L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. 2

[42] J. VanderPlas, B. E. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of open source software*, 3(32):1057, 2018. doi: 10.21105/joss.01057 2

[43] J. Vig. A multiscale visualization of attention in the transformer model. *CoRR*, abs/1906.05714, 2019. doi: 10.48550/arXiv.1906.05714 2

[44] J. Wang, L. Gou, H. Yang, and H.-W. Shen. GANViz: A visual analytics approach to understand the adversarial game. *IEEE Transactions on Visualization and Computer Graphics*, 24(6):1905–1917, 2018. doi: 10.1109/TVCG.2018.2816223 1, 2, 3

[45] Z. J. Wang, R. Turko, and D. H. Chau. Dodrio: Exploring transformer models with interactive visualization. *CoRR*, abs/2103.14625, 2021. doi: 10.48550/arXiv.2103.14625 2, 3

[46] Z. J. Wang, R. Turko, O. Shaikh, H. Park, N. Das, F. Hohman, M. Kahng, and D. H. Polo Chau. CNN explainer: Learning convolutional neural networks with interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1396–1406, 2021. doi: 10.1109/TVCG.2020.3030418 2, 3

[47] H. Wickham. ggplot2. *WIREs Computational Statistics*, 3(2):180–185, 2011. doi: 10.1002/wics.147 2

[48] X. Xuan, X. Zhang, O.-H. Kwon, and K.-L. Ma. VAC-CNN: A visual analytics system for comparative studies of deep convolutional neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 28(6):2326–2337, 2022. doi: 10.1109/TVCG.2022.3165347 3

[49] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. *CoRR*, abs/1506.06579, 2015. doi: 10.48550/arXiv.1506.06579 1, 2

[50] J. Zhao, Z. Dai, P. Xu, and L. Ren. ProtoViewer: Visual interpretation and diagnostics of deep neural networks with factorized prototypes. In *Proceedings of IEEE Visualization Conference*, pp. 286–290. IEEE, Salt Lake City, 2020. doi: 10.1109/VIS47514.2020.00064 3