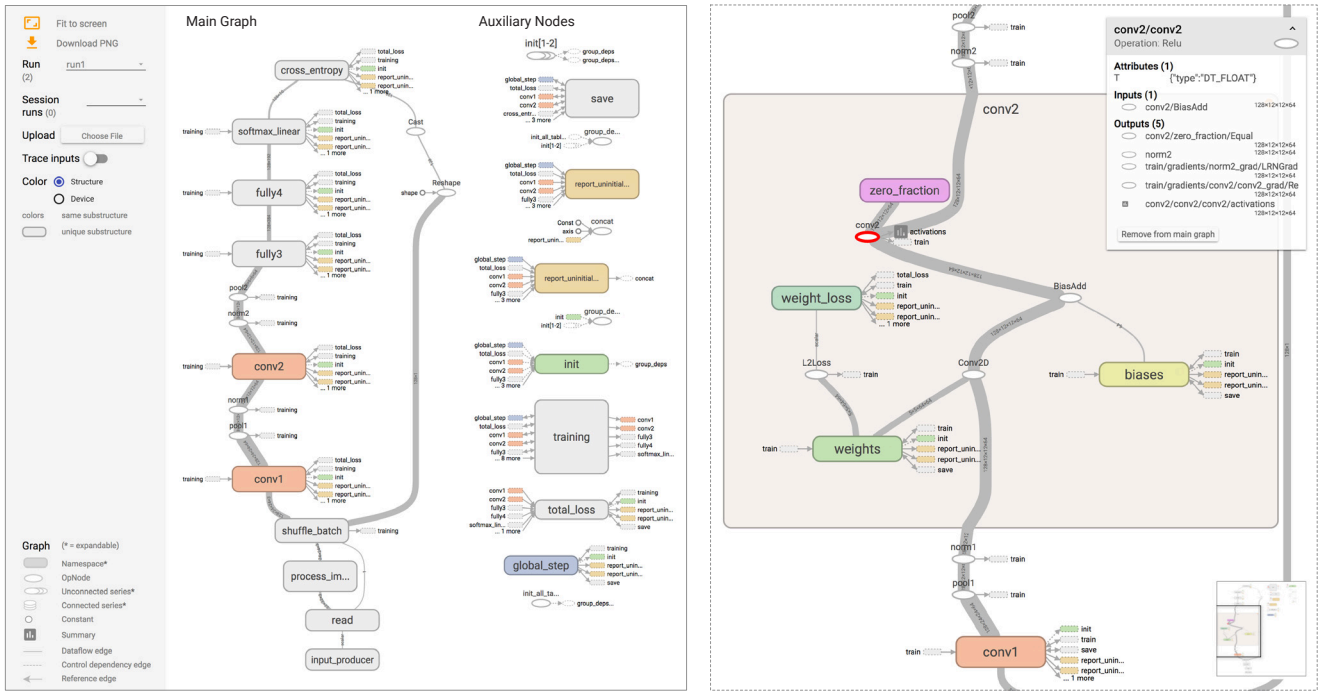
Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow

Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson,
Dandelion Mané, Doug Fritz, Dilip Krishnan, Fernanda B. Viégas, and Martin Wattenberg



(a) (b)

Fig. 1. The TensorFlow Graph Visualizer shows a convolutional network for classifying images (`tf_cifar`) . (a) An overview displays a dataflow between groups of operations, with *auxiliary nodes* extracted to the side. (b) Expanding a group shows its nested structure.

Abstract—We present a design study of the TensorFlow Graph Visualizer, part of the TensorFlow machine intelligence platform. This tool helps users understand complex machine learning architectures by visualizing their underlying dataflow graphs. The tool works by applying a series of graph transformations that enable standard layout techniques to produce a legible interactive diagram. To declutter the graph, we decouple non-critical nodes from the layout. To provide an overview, we build a clustered graph using the hierarchical structure annotated in the source code. To support exploration of nested structure on demand, we perform edge bundling to enable stable and responsive cluster expansion. Finally, we detect and highlight repeated structures to emphasize a model's modular composition. To demonstrate the utility of the visualizer, we describe example usage scenarios and report user feedback. Overall, users find the visualizer useful for understanding, debugging, and sharing the structures of their models.

Index Terms—Neural Network, Graph Visualization, Dataflow Graph, Clustered Graph

1 Introduction

Recent years have seen a series of breakthroughs in machine learning, with a technique known as *deep learning* bringing dramatic results on standard benchmarks [37]. A hallmark of deep learning methods is their multi-layered networks of calculations. The complexity of these networks, which often include dozens of layers and millions of parameters, can lead to difficulties in implementation. Modern deep learning platforms including TensorFlow [6], Theano [11], and Torch [18] provide high-level APIs to lower these difficulties. With these APIs, developers can write an abstract program to generate a low-level dataflow graph that supports a variety of learning algorithms, distributed computation, and different kinds of devices.

These APIs and their dataflow models simplify the creation of neural networks for deep learning. Yet developers still have to read code and manually build a mental map of a model to understand its complicated structure. A visualization of the model can help developers inspect its structure directly. However, these dataflow graphs typically contain thousands of heterogeneous, low-level operations; some of which are high-degree nodes that connect to many parts of the graphs. As a result, standard layout techniques such as flow layout [49] and

- *Kanit Wongsuphasawat is with Paul G. Allen School of Computer Science & Engineering, University of Washington. E-mail: kanitw@uw.edu.*
- *Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mané, Doug Fritz, Dilip Krishnan, Fernanda B. Viégas, and Martin Wattenberg are with Google Research. E-mail: {smilkov, jwexler, jimbo, dougfritz, dilipkay, viegas, wattenberg}@google.com*

Manuscript received 31 Mar. 2017; accepted 1 Aug. 2017.
Date of publication 28 Aug. 2017; date of current version 1 Oct. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TVCG.2017.2744878

force-directed layout generally produce tangled diagrams.

In response, we present the TensorFlow Graph Visualizer, a component of in the TensorFlow machine intelligence platform, to help developers understand and inspect the structure of their TensorFlow models. Given a low-level directed dataflow graph of a model as input, the visualizer produces an interactive visualization that shows the high-level structure of the model, akin to diagrams that deep learning experts typically draw to explain their models, and enables users to explore its nested structure on demand.

This paper describes our design process and the design of the visualizer. We present a set of graph transformations and visual encodings that enables standard flow layout techniques [50] to produce a legible interactive diagram from a dataflow graph of a machine learning model. To provide an overview, we build a clustered graph by grouping nodes based on their hierarchical namespaces that developers can provide. To support exploration, we introduce a novel application of edge bundling to enable stable and responsive expansion of clustered flow layout. To declutter the graph, we apply heuristics to extract non-critical nodes, and introduce new visual encodings that decouple the extracted nodes from the layout. We also detect and highlight repeated structures to help users recognize modular composition in the models. Finally, we overlay the graph with additional quantitative information to help developers inspect their models.

To demonstrate the utility of the visualizer, we describe usage scenarios for exploring deep learning models. We also report feedback from users who use the tool to examine structures of deep learning models, and discuss usage patterns we have observed in the wild. Overall, developers find the visualization useful for understanding, debugging, and sharing the structures of their models.

2 Related Work

2.1 Dataflow Graph Applications

Dataflow graphs arise in diverse domains: distributed systems [15, 32, 42], databases [41], user-interface development [20], visualization [14, 46, 52] and engineering [4].

Some dataflow systems (*e.g.,* [4, 14, 52]) use visualizations as authoring tools and allow users to directly edit the graph to modify the dataflow. Since dataflows in these systems represent high-level components that are manually added one-by-one, their graphs are typically much smaller compared to dataflow graphs of deep neural networks.

One important application domain for dataflow models is large-scale distributed systems, which automatically create dataflow structures from a program to enable distributed computation [15, 32, 42]. To help users diagnose performance of distributed databases, Perfopticon [41] includes a collapsible dataflow graph of query execution plans. However, its design does not scale to large dataflow graphs.

Our goal is to help users understand large, complex dataflow programs with a visualization that scales, provides a legible overview, and supports detailed exploration on demand. While our design targets TensorFlow, our strategy to decouple non-critical nodes from the layout can be applicable for heterogeneous graphs in other domains. Clustered flow graphs in other domains can apply edge bundling to facilitate responsive and stable graph expansion as well.

2.2 Visualization for Neural Networks

Visualization plays many important roles in machine learning. Practitioners and researchers often use visualization to monitor learned parameters and output metrics to help them train and optimize their models. Besides the Graph Visualizer, TensorBoard, TensorFlow's dashboard component, also includes modules for monitoring scalar values, distribution of tensors, images, and audio. We briefly describe these modules in supplementary material.

For neural networks, the lack of understanding of how the models work often makes model optimization difficult. Visualizations can improve the transparency and interpretability of the models and help open these "black boxes" [34, 54]. Some projects present visualizations for specific types of neural networks such as convolutional network [39] and recurrent networks [2, 48]. Besides supporting expert's analysis, recent projects, such as Olah's interactive essays [1], ConvNetJS [2], and TensorFlow Playground [47], provide interactive visualizations to teach novices how neural networks work. Unlike prior projects that focus on visualizing learned parameters and output values, or specific kinds of networks, our primary goal is to help users understand the structure of dataflow graphs that represent arbitrary neural networks.

Similar to our work, some high-level deep learning toolkits such as Keras [3] and MXNet [16] leverage GraphViz [28] to provide tools for visualizing model structure. However, their graph representations are higher-level than TensorFlow and do not contain information about nested low-level operations. For other toolkits that use more complex and lower-level dataflow graphs [11, 18], standard tools like GraphViz generally produce illegible layouts. Without a better tool, developers have to read the code and manually build a mental map of the structure to understand a model. Our visualizer aims to help developers understand and inspect low-level dataflow graphs of neural networks.

2.3 Graph Visualization Techniques

Visualizing dataflow graphs can be generalized as drawing directed graphs. A common way to draw directed graph is the flow layout, which uses one axis to convey overall direction. A standard flow layout algorithm, introduced by Sugiyama et al. [50] and widely extended [13, 31, 33], assigns x-and y-coordinates separately in multiple stages that optimize different objectives. An alternative approach by Dwyer et al. applies constraint optimization to compute both coordinates for a flow layout simultaneously [22, 23, 24]. The separation constraints [24] introduced by this approach are also used in other types of layouts [35, 57]. Since directionality is critical for understanding the dataflow of neural networks, we use a flow layout as a basis of our layout algorithm and augment it with additional transformations to simplify the graph. Our implementation uses a Sugiyama-style algorithm due to the availability of stable libraries and high-quality documentation. However, our additional transformations are also applicable for a constraint-based flow layout.

To simplify large graphs, a common technique is to build hierarchical clustered graphs that provide an overview and support cluster expansion to explore details [7, 8, 9, 30]. Following this approach, we leverage hierarchical namespaces that developers provide to create a clustered flow layout. To help users maintain mental maps during exploration [40], a clustered graph must also be responsive and stable. For undirected graphs, some systems use static layouts [10, 30, 55] while others draw the graph interactively [7, 8, 55]. For directed graphs, constraint-based approaches [24, 25, 26] and an online technique [43] can speed up the calculation and preserve the topology of the flow layout during interactions. However, drawing edges directly between all visible nodes still clutters expanded graphs. To both declutter the view and support interactive expansion, we bundle and route edges between groups such that edges only connect nodes that are siblings in the hierarchy [10]. This approach allows us to compute the layout of each cluster's subgraph separately and update only relevant subgraphs when users expand nodes. As a result, we can create responsive and stable interactive graph with a standard Sugiyama-style algorithm for clustered graph [27, 29, 45, 49], without the need to implement complex constraints or online algorithms. To the best of our knowledge, none of the prior work documents the application of edge bundling to enable interactive expansion of a clustered flow layout.

Graph clustering and edge bundling simplify our layout, but still leave intertwining edges in many cases due to the presence of non-critical, high-degree nodes. Another group of graph simplification techniques extracts nodes and edges, or replaces them with special visual encodings. Dunne and Shneiderman substitute common nodes and links with compact glyphs that represent common patterns [21]. Van Ham & Wattenberg remove edges based on a centrality measure to show graph clusters in force-directed layout [56]. Inspired by these strategies, we apply heuristics based on domain knowledge, semantic metadata, and distribution of node degrees to extract non-critical nodes from the layout, and re-encode them with new visual encodings.

3 Background: TensorFlow

TensorFlow [6] is Google's system for the implementation and deployment of large-scale machine learning models. Although deep learning is a central application, TensorFlow also supports a broad range of models including other types of learning algorithms.

The Structure of a TensorFlow Model

A TensorFlow model is a dataflow graph that represents a computation. Nodes in the graph represent various *operations*. These include mathematical functions such as addition and matrix multiplication; constant, sequence, and random operations for initializing tensor values; summary operations for producing log events for debugging; and variable operations for storing model parameters.

Edges in TensorFlow graphs serve three different purposes. *Data dependency* edges represent *tensors*, or multidimensional arrays, that are input and output data of the operations. *Reference* edges, or outputs of variable operations, represent pointers to the variable rather than its value, allowing dependent operations (*e.g.,* `Assign`) to mutate the referenced variable. Finally, *control dependency* edges do not represent any data but indicate that their source operations must execute before their tail operations can start.

Building a TensorFlow Model

The TensorFlow API provides high-level methods for producing low-level operations in the dataflow graph. Some, such as `tf.train.GradientDescentOptimizer`, generate dozens of low-level operations. Thus a small amount of code, such as the definition of the **`tf_mnist_simple`** model in Figure 4 (see supplementary material), can produce about a hundred operations in the graph. Real-world networks can be even more complex. For instance, an implementation of the well-known Inception network [51] has over 36,000 nodes.

Operation Names

For clarity, operations in TensorFlow graphs have unique names, which are partly generated by the API and partly specified by users. Slashes in operation names define hierarchies akin to Unix paths (`like/this/example`). By default, the API uses operation types as names and appends integer suffixes to make names unique (*e.g.,* "`Add_1`"). To provide a meaningful structure, users can group operations with a namespace prefix (*e.g.,* "`weights/`"). Complex methods such as `tf.train.GradientDescentOptimizer` also automatically group their underlying operations into subnamespaces (*e.g.,* "`gradients`" and "`GradientDescent`"). As discussed later in §5.2, we apply these namespaces to build a clustered graph.

4 Motivation & Design Process

The motivation for the TensorFlow Graph Visualizer came from our conversations with deep learning experts, including one of the authors. When experts discuss their models, they frequently use diagrams (*e.g.,* Figure 2) to depict high-level structure. When working with a new model, they often read the code and draw a diagram to help them build a mental map of the model's architecture. Since diagrams are critical for their work, machine learning experts desire a tool that can automatically visualize the model structures.

Motivated by an apparent need for a visualization tool, we worked with potential users to understand the key tasks for such a visualization. We also examined the model data that a visualization would have to portray. The purpose of these investigations was to match user needs with what would be possible given real-world data.

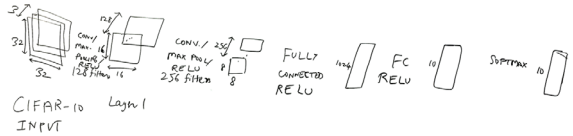


Fig. 2. Whiteboard drawing by a computer vision expert: a convolutional network for image classification.

4.1 Task Analysis

Our overarching design goal is to help developers understand and communicate the structures of TensorFlow models, which can be useful in many scenarios. Based on conversations with potential users, we identified a set of key scenarios for a model visualization. Beginners often learn how to use TensorFlow based on example networks in the tutorials. Even experts usually build new models based on existing networks. Therefore, they can use the visualization to help them understand existing models. When modifying the code that generates models, developers can use the visualization to observe the changes they make. Finally, developers typically work in teams and share their models with their co-workers; they can use the visualization to help explain the structures of their models.

As discussed earlier, researchers often manually create diagrams to explain their models and build mental maps. These diagrams were an important inspiration for our design. As shown in Figure 2, they usually show a high-level view of the model architecture and feature relatively few nodes. Low-level implementation details such as cost function calculation or parameter update mechanism are generally excluded from these diagrams. When a network features repeated modules, the modules are usually drawn in a way that viewers can tell they are the same. These diagrams also often annotate quantitative information such as the layer dimensions in Figure 2.

In model development, developers also need to understand the model beyond the high-level structure. For example, when a developer modifies a part of the code and sees an unexpected result, the reason may lie either in the model itself or in the code that created the model. It can be hard to know whether the program is actually building the intended model. In such case, the developer may desire to inspect a specific part of the model to debug their code.

From conversations with experts about potential usage scenarios and our observation from these hand-drawn diagrams, we identify a set of main tasks that the visualization should support:

- T1: Understand an **overview** of the high-level components of the model and their relationships, similar to schematic diagrams that developers manually draw to explain their model structure.
- T2: Recognize **similarities and differences** between components in the graph. Knowing that two high-level components have identical structure helps users build a mental model of a network; noticing differences between components that should be identical can help them detect a bug.
- T3: Examine the **nested structure** of a high-level component, in terms of low-level operations. This is especially important when a complex nested structure has been created automatically from a single API call.
- T4: Inspect **details** of individual operations. Developers should not have to refer back to the code to see lists of attributes, inputs, and outputs, for instance.
- T5: View **quantitative data** in the context of the graph. For example, users often want to know tensor sizes, distribution of computation between devices, and compute time for each operation.

These tasks do not include standard monitoring apparatus, such as plots of loss functions (*i.e.* optimization objectives) over time. Such tools are important, but beyond the scope of this paper since they do not relate directly to the structure of the dataflow graph; we briefly discuss how TensorFlow handles these issues in supplementary material.

This task analysis guided our work, as we engaged in a user-centered design process. Throughout the project, we worked closely with several TensorFlow developers and beta users and iterated on the design. We met with beta users and members of the developer team for at least an hour a week (sometimes for much longer) for about 20 weeks. After the release, we continued to seek feedback from both internal and public users.

4.2 Data Characteristic Analysis

Our design process also included an investigation into the particular properties of dataflow graphs that define TensorFlow models. An immediate concern was that early experiments with standard layout tools

(e.g. flow layout in GraphViz [28], as well as force-directed layouts from D3) had produced poor results. We wanted to understand, more generally, the scale of real-world model data, and whether it would be possible for an automatic visualization to support key user tasks.

We initially performed rapid prototyping to investigate the reasons that TensorFlow graphs caused problems for standard layouts. We visualized several example computation model graphs in multiple ways. After a few trials, we quickly abandoned experiments with force-directed layouts as they created illegible hairballs. Attempts to use a standard flow layout [50] for example models yielded illegible results. For example, Figure 4-a shows a flow layout of a simple network for classifying handwritten digits (**`tf_mnist_simple`**). Building clustered flow layouts allows us to produce more legible views. However, these layouts were still cluttered and often change dramatically after expanding a node.

These experiments pointed to several challenges that make TensorFlow model graphs problematic for standard techniques.

C1: **Mismatch between graph topology and semantics.** One might hope that meaningful structures would emerge naturally from the graph topology. Unfortunately, it is hard to observe clear boundaries between groups of operations that perform different functions such as declaring and initializing variables, or calculating gradients. Moreover, randomized algorithms produce visually different layouts for topologically identical subgraphs. A good layout should show similarities between identical subgraphs.

C2: **Graph heterogeneity.** Some operations and connections are less important for understanding the graph than others. For example, developers often consider constants and variables simply as parameters for other operators. Similarly, summary operations serve as bookkeepers that save their input to log files for inspection, but have no effect on the computation. Treating all nodes equivalently clutters the layout with non-critical details.

C3: **Interconnected nodes.** While most nodes in TensorFlow graphs have low-degree (one to four), most graphs also contain some *interconnected* high-degree nodes that couple different parts of the graphs. For example, the summary writer operation (Figure 4-a) connects with all summary operations. These high-degree nodes present a major problem for visualizations, forcing a choice between tight clustering and visual clutter. In force-directed layouts, connections between these nodes reduce distances between nodes that are otherwise distant, leading to illegibly dense groupings. In flow layouts, these connections produce long edges along of the flow of the layout and clutter the views.

5 Design of The TensorFlow Graph Visualizer

We now describe the design of TensorFlow Graph Visualizer that aims to help users with the tasks in §4.1. First, we explain the basic layout and visual encodings. We then describe a sequence of graph transformations that target the challenges in §4.2. We also report how we identify and highlight repeated structure, and overlay the graph with other quantitative information. We finally discuss our implementation.

For simplicity, we will use a simple softmax regression model for image classification (**`tf_mnist_simple`**) to illustrate how we transform an illegible graph into a high-level diagram (Figure 4). As shown in the final diagram (Figure 4-d), the model calculates weighted sum (`Wx_b`) of the input `x-data`. The parameters (`weights` and `bias`) are placed on the side. With `Wx_b` and the `y-input` labels, the model computes the `test` metrics and cross-entropy (`xent`), which is in turn used to `train` and update the parameters. Besides this simple model, we describe scenarios for exploring more complex neural networks in §6.

5.1 Basic Encoding and Interaction

As in Figures 1 and 4-d, the visualizer initially fits the whole graph to the screen. We draw the directed graph of the dataflow with a flow layout [50] from the bottom to the top, following a common convention in the deep learning literature. Although both horizontal and vertical layouts are common in the literature, we use a vertical layout since it produces a better aspect ratio for models with a large number of layers.

Horizontal ellipse nodes represent individual operations. Compared to a circle, this shape provides extra room for input edges on the bottom and output edges on the top. Edge styles distinguish different kinds of dependencies (T2). Solid lines represent data that flow along *data dependency* edges. Dotted lines indicate that data does not flow along *control dependency* edges (*e.g.,* `bias`→`init` in Figure 4-c). *Reference* edges, such as `weight`→`weight/Assign` in Figure 3-a, have arrows pointing back to the variables to suggest that the tail operations can mutate the incoming tensors.

Since users are often interested in the shape of tensors edges (T5), we label the edges with the tensor shape (Figure 3-a). We also compute the total *tensor size* (*i.e.* the number of entries, or the product of each dimension size) and map it to the edge's stroke width (via a power scale due to the large range of possible tensor sizes). If the input graph does not specify dimension sizes for a tensor, the lowest possible tensor size determines the width.

Users can pan and zoom to navigate the graph by dragging and scrolling. When the graph is zoomed, a mini-map appears on the bottom right corner to help users track the current viewpoint (Figure 1-b). To reset the navigation, users can click the "Fit to screen" button on the top-left. To inspect details of a node (T4), users can select a node to open an information card widget (Figure 1-b, top-right), which lists the node's type, attributes, as well as its inputs and outputs. The widget itself is interactive: users can select the node's input or output listed on the graph. If necessary, the viewpoint automatically pans so the selected node is visible.

5.2 Graph Transformation

The key challenge for visualizing TensorFlow graphs is overcoming the layout problems described in §4.2. We apply a sequence of transformations (Figure 4) that enables standard layout techniques to overcome these challenges. To provide an overview that matches the semantics of the model (C1), we cluster nodes based on their namespaces [8]. In addition to clustering, we also bundle edges to enable stable and responsive layout when users expand clusters. Moreover, as non-critical (C2) and interconnected nodes (C3) often clutter the layout of clustered graphs, we extract these nodes from the graphs and introduce new visual encodings that decouple them from the layout.

Step 1. Extract Non-Critical Operations

TensorFlow graphs are large and contain heterogeneuous operations (C2), many of which are less important when developers inspect a model. To declutter and shrink the graph, we de-emphasize these non-critical operations by extracting these nodes from the layout and encoding them as small icons on the side of their neighbors.

We extract two kinds of operations, *constants* and *summaries*. Both are *loosely connected*: extracting them does not change any paths between other nodes. A constant node always serves as an input to another operation and thus has only one output edge and no input edge. A summary node always has one input edge from a logged operation and one output edge to the summary writer node, which is the global sink node that takes all summary nodes as inputs and write log data to the log file. Since the summary writer behaves identically in every TensorFlow graph, it is negligible for distinguishing different models and can be removed. With the summary writer removed, both summaries and constants have degree 1. Thus, we can extract them without changing any connections between the rest of the graph.

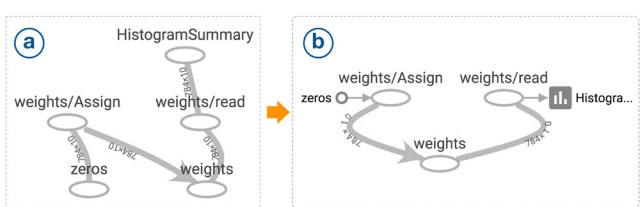


Fig. 3. Extract non-critical operations. (a) A raw subgraph for the `weights` variable and its summary. (b) The subgraph with summary and constant operations extracted from the flow layout and re-encoded as embedded input and output on the side of their neighbors.

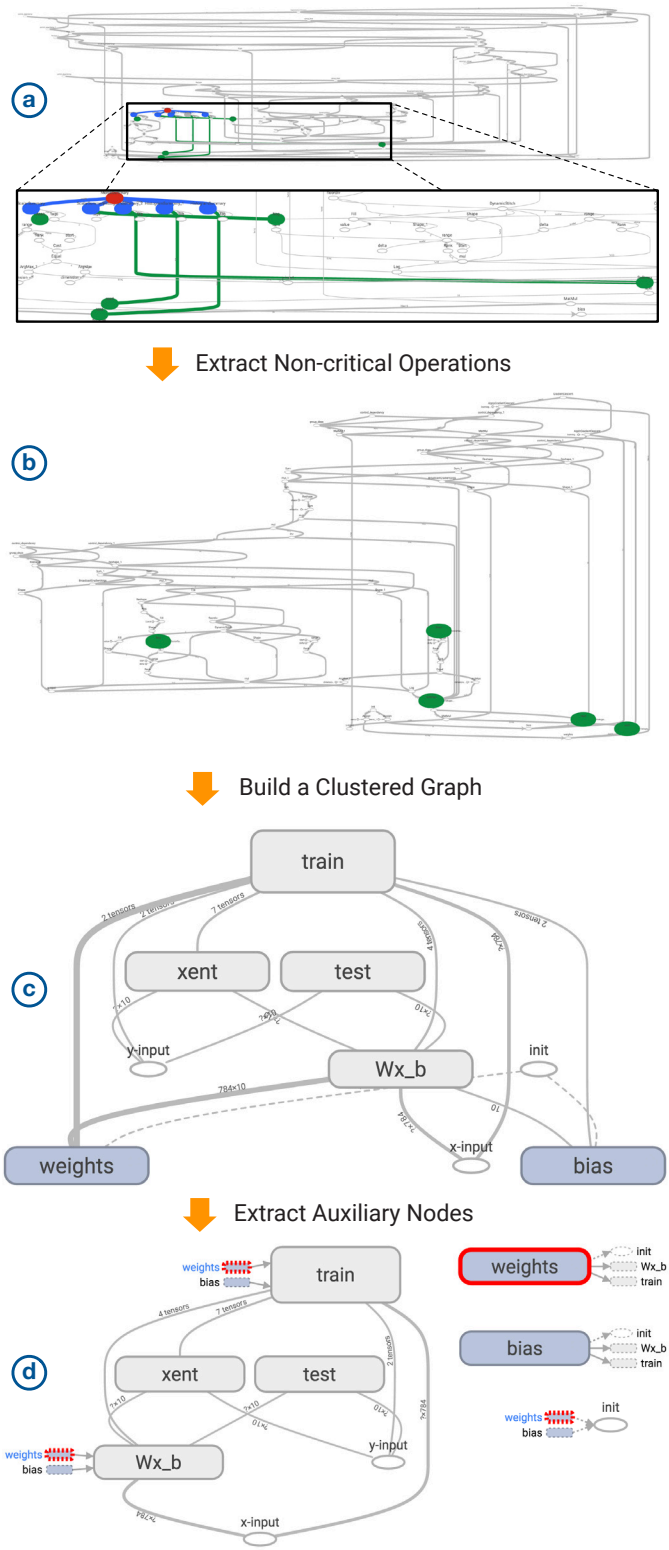


Fig. 4. Transforming the graph of a simple model for classifying handwritten digits (`tf_mnist_simple`). (a) A dataflow graph, which is large and wide and has many intertwining connections. The zoomed part of the raw graph highlights how the summary writer operation (red) is interconnected to logged operations in many different parts of the graph (green) via summary operations (blue). (b) The dataflow graph with summary and constant operations extracted. Logged operations (green) are now less intertwined. (c) An overview of the graph, which shows only top-level group nodes in the hierarchy. (d) An overview with auxiliary nodes extracted to the side of the graph. Selecting an extracted node highlights *proxy icons* attached to its graph neighbors.

We encode the extracted constants and summaries as *embedded inputs and outputs* of their neighbor operations, or small icons on the left and right of the node they connect to (Figure 3). A small circle represents a constant while a bar chart icon represents a summary operation. Edges of the embedded nodes have arrows to indicate the flow direction. We place embedded nodes on the side of their neighbor nodes to make the overall layout more compact and avoid occlusion with other edges that connect with the node on the top and bottom.

As shown in Figure 4 (a-b), this transformation declutters the view in two ways. First, removing the interconnected summary writer (red) frees logged operations (green) from being tied together (C3). Moreover, constants and summaries together account for a large fraction of nodes (approximately 30% in a typical network). Extracting them can significantly reduce the graph size, making it less packed. Reduced size also expedites subsequent transformation and layout calculation.

Step 2. Build a Clustered Graph with Bundled Edges

To reflect semantic structure of the model (C1), we build a hierarchical clustered graph by grouping operations based on their namespaces. We also bundle edges between groups to help simplify the layout and make the clustered flow layout responsive and stable when users expand nodes. With these transformations, we can provide an overview (T1) that shows only the top-level groups in the hierarchy as the initial view, while allowing users to expand these groups to examine their nested structure (T3).

Figure 4-c shows an example overview produced from this step. Each rounded rectangle represents a group node. To distinguish groups of different size (T2), each rectangle's height encodes the number of operations inside the group. Users can expand these groups to examine their nested structure, as shown in Figure 5-c.

Building a hierarchy. We build a hierarchy based on operation names by creating a *group node* for each unique namespace (or, in the Unix path analogy, directory). If a node's name conflicts with a namespace (analogy: a file having the same name as a directory in Unix) we put the node inside the group node and add parentheses around its name. For example, Figure 5-a shows a hierarchy, which groups three nodes in Figure 3-b under `weights`. The original `weights` operation becomes the `(weights)` operation inside the `weights` group.

Although namespace groupings are optional, they are a good choice for defining a hierarchy for a few reasons. First, TensorFlow graphs typically have informative namespaces as developers also use these namespaces to understand non-visual output in the debugging tools. Moreover, inferring the semantic hierarchy from the graph topology alone is ineffective; even incomplete namespace information better corresponds to the mental model of the network's developer. Most importantly, adding names to models is relatively low effort for developers; we predicted that they would add namespaces to their models to produce hierarchical groupings in the visualization if necessary. As described later in §7, user feedback confirms this prediction.

To prevent operations without proper namespace prefixes (*e.g.,* "Add_1", "Add_2", ...) from cluttering the view, we also group operations of the same name under the same namespace into a special *series node*. To avoid aggressive series grouping, by default we only group these operations if there are at least five of them.

Bundling edges to enable stable and responsive expansion. After grouping nodes to build a hierarchy, we draw *group edges*, or edges between groups. We avoid drawing edges between all visible nodes directly for a few reasons. First, it usually produces cluttered layouts. More importantly, it may require complete layout re-calculation and cause major layout changes every time the user expands a node. Instead, we bundle edges and route them along the hierarchy to make the layout responsive, stable, and legible.

To do so, we create group edges only between nodes within the same subgraphs of the group nodes. Within a subgraph, we create group edges between group nodes and operation nodes as well as group edges between pairs of group nodes. A group node and an operation are dependent if there is at least one edge between a descendant operation of the group node and the operation. Similarly, two group nodes are dependent if there is at least one depen-

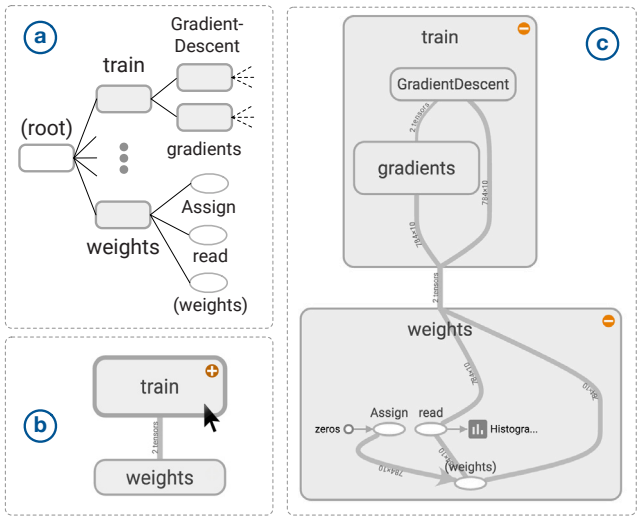


Fig. 5. Build a hierarchical clustered graph. (a) A hierarchy showing only `train` and `weights` namespaces from **`tf_mnist_simple`** in Figure 4. (b) A high-level diagram showing dependency between `train` and `weights`. Hovering over the `train` namespace shows a button for expansion. (c) A diagram with `train` and `weights` expanded.

dency edge between a pair of their descendant operations. If there is more than one such edge, the corresponding group edge can bundle multiple dependency edges. For example, we create a group edge from `weights` to `train` in Figure 5-b. This group edge actually bundles two dependency edges: `weights/read` → `train/gradients` and `weights/(weights)` → `train/GradientDescent`.

When group nodes are expanded, we *route* edges along the hierarchy instead of directly drawing edges between all visible nodes [10]. We only directly draw edges between nodes that are siblings in the hierarchy. For an edge between non-sibling nodes, we split the edge into segments that are routed through their siblings ancestors. For example, in Figure 5-c, both `weights/read` → `train/gradients` and `weights/(weights)` → `train/GradientDescent` are routed through `weights` and `train`. Routed edge bundling provides multiple benefits:

1. *Responsiveness.* With edge routing, we can compute the layout of each group node's subgraph separately because edges do not cross the boundary of each group node. Since layered graph layout has super-linear time complexity, dividing the layout calculation into smaller subproblems can provide significant speed-up. Moreover, when a group node is expanded or collapsed, we only need to re-compute the layout for the ancestors of the group node instead of recomputing the whole layout since other nodes are unaffected. In contrast, if we directly draw edges between all visible nodes even though they are not a part of the same group node's subgraph, the layout must be computed all at once. The whole layout also has to be recomputed every time a node is expanded or collapsed.

2. *Stability.* With edge routing, the topology of each group node's subgraph remains constant after an expansion. Therefore, expanding a node only enlarges the node and its ancestors without causing major changes to the layout. This helps users maintain their mental model of the graph when they interactively expand the graph. We chose not to directly draw edges between all visible nodes, since in that case an expansion could vastly change the graph's topology and produce a totally different layout.

3. *Legibility.* Edge routing decreases the number of edges in each group's subgraph and thus declutters the layout by reducing edge crossings. Drawing edges directly can tangle the view with many crossing curves, especially when many nodes are expanded.

One possible drawback of edge routing is that it can be harder to trace where an edge goes when it is bundled. We address this with interaction by providing a list of inputs and outputs in the information card to help them track a particular input and output edge.

Step 3. Extract Auxiliary Nodes from the Clustered Graph

Building a clustered graph simplifies the layout and provides a high-level overview. However, high-degree nodes with connections across the graph continue to present a challenge, causing intertwining edges that clutter the visualization (C3). Ironically, when we showed these diagrams to experts, they commented that many problematic nodes such as variables and bookkeeping operations are actually not important for understanding the model structure.

Akin to Step 1, we address these issues by extracting non-critical nodes, but from each subgraph in the clustered graph instead of the raw input graph. We place extracted nodes on the right of the layout (labeled as "auxiliary nodes") as shown in Figures 1-a and 4-d. To represent the extracted connections, we add small *proxy icons* for them as embedded inputs and outputs besides their neighbor nodes. Each proxy icon has a shape like its node (rectangles for groups and ellipses for operations) and has a dashed border to indicate that it serves as a proxy for its full representation on the side. When an extracted node or one of its proxies is selected or hovered, we highlight both the node and all of its proxies to help users locate the node and its neighbors. This extraction strategy enables us to simplify the graph, while retaining connectivity information from the original graph.

The key challenge here is to determine non-critical nodes to extract. We use a set of heuristics to extract two kinds of auxiliary nodes for each subgraph. First, we extract auxiliary nodes with specific and consistent subgraph patterns. These include groups for declaring and initializing variables, which experts consider as parameters rather than core operations, and NoOp nodes, which serve as control dependency hubs that perform no computation.

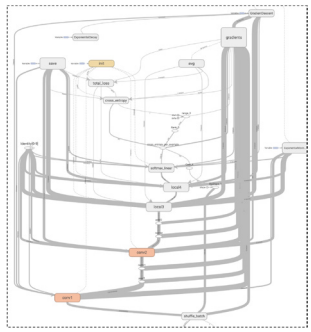


Fig. 6. Before extracting auxiliary nodes, the links between groups clutter the overview of **`tf_cifar`** in Figure 1-a.

We then extract auxiliary nodes that do not have specific subgraph patterns, but connect to many parts of the graph. For example, groups that compute statistics for measuring performance often connect to multiple layers in a network. Due to their high-degree, these auxiliary nodes are mainly the cause of the intertwining connections (C3). Meanwhile, core computations in TensorFlow are typically mathematical operations that are binary or ternary operators. Thus most core nodes have lower degree than auxiliary nodes, except the ones that connect to many auxiliary nodes. Most of these auxiliary nodes are also *sink-like*, appearing at or near the end of the graph, and thus have high in-degree. Since extracting nodes affects the degrees of their neighbors, we extract high in-degree nodes before high out-degree nodes so core nodes that connect to many sink-like auxiliary nodes no longer have high out-degree.

To extract high in-degree nodes, we first calculate the quartiles of in-degrees for a subgraph, ignoring edges of extracted nodes. We apply Tukey's method [53] to detect outlier nodes with in-degrees higher than $Q3 + k * (Q3 - Q1)$, where $k = 1$ (a slightly aggressive threshold). To avoid extracting nodes in low-degree graphs, we only extract the outliers if they have in-degrees higher than a minimum threshold of 4. (To demonstrate our transformations with a simplified example, we disable this threshold in Figure 4.) After extracting high in-degree nodes, we repeat the same procedure to extract out-degree nodes, but use a conservative threshold ($k = 4$) to avoid extracting core nodes.

Another subtlety for calculating in- and out-degree in TensorFlow is that data dependency edges are considered more important than control dependency edges. If a node contains only a few data edges but many control edges, the node is likely a core operation. On the other hand, if a node contains only control edges, the node is likely auxiliary. Thus, if a node has a data edge, we determine the node's degree by only the number of data edges. If it has only control edges, we determine its degree by the number of control edges. As a result, we can further distinguish between core and auxiliary nodes.

As shown in Figures 1 and 4-d, extracting auxiliary nodes declutters clustered graphs and provides a layout that shows core structures of the models. Since we use heuristics to extract nodes, we also allow users to override the heuristics; users can use the "Remove from / Add to the main graph" button in the information card (Figure 1-b, right) to manually extract a node or re-attach a node back to the main layout.

5.3 Identify and Highlight Repeated Structure

The use of repeated modules is a characteristic feature of deep learning networks. For example, many image classification systems [51] have a series of identical convolutional layers. The *structure view* (our default view, as shown in Figures 1 and 4-d) helps users understand a complex dataflow by highlighting group nodes with identical structure with the same color (T2). Uniquely structured group nodes are otherwise gray.

The challenge in creating this view is that, a priori, there is no explicit marker for identical structures. In theory one could modify the APIs to include structural annotations along with node names. However, keeping structural annotations in sync with the actual structure would require error-prone manual effort during changes to the APIs.

Instead, the TensorFlow Graph Visualizer automatically looks for repeated substructures in the graph. Since detecting isomorphic subgraphs is an NP-complete problem [19], we use a heuristic approach. Although this strategy is not guaranteed to find arbitrary repeated structures, it performs well and is effective in practice.

Detecting structural templates from group nodes. We restrict our search to subgraphs that are defined by single group nodes. This restriction is reasonable in practice, since repeated modules are frequently created by calling a given function multiple times, creating groups with similar nested structures. We detect similar group nodes using a two-step approach inspired by the *blocking* technique for duplication detection in databases [17]. This technique reduces the number of record pairs to be compared by splitting the entities into blocks such that only entities in the same block need to be compared. Here, we first create a hashmap to store clusters of nodes with the same *blocking key* based on metadata that we compute while building the hierarchy (§5.2). The key consists of the number of nodes and edges, and a histogram counting types of operation nodes inside the group node's subgraph. Two identical subgraphs will have the same key since all of these properties are isomorphism invariants.

Next, we examine each cluster in the hashmap to find templates of repeated modules, starting from clusters that contain fewer operations. For each cluster c, we first initialize a set of templates to an empty set $T_c = \{\}$. For each group node g in the cluster, we compare it with each existing template $t \in T_c$. For each template t, we compare g with a node g_t that belongs to t using a subgraph similarity checking method described in the next paragraph. If g and g_t are similar, we assign g to the template t. Otherwise, we continue checking with other templates. If g does not match any existing templates, we add a new template with g as a member to the set T_c. After visiting all nodes in the cluster, we add all templates $t \in T_c$ that have more than one member to the global template set T, which is used for assigning colors.

Checking subgraph similarity. We determine if two group nodes g_1 and g_2 are similar with the following steps. If their subgraphs s_1 and s_2 do not have the same *degree sequence*, they are dissimilar. Otherwise, we use a heuristic graph traversal procedure that determines graph similarity with nodes' signature. We define a node's *signature* as a tuple of (1) the node's type, which can be an operation type for an operation node or its template unique identifier[1] for a group node, (2) the node's in-degree, and (3) the node's out-degree. We traverse through both graphs using breadth first search. First, we add the source nodes of the subgraphs s_1 and s_2 to their queues q_1 and q_2 respectively. If there are multiple sources for each subgraph, we sort them by their signatures before adding to the queue. We then traverse by dequeueing a pair of nodes, one from each queue. For each pair of visited nodes (n_1, n_2), we compare their signatures. If they are different, we can terminate the process and decide that the group nodes are dissimilar. Otherwise, we add all direct successors of n_1 and n_2 to q_1 and q_2 respectively. If there are multiple successors, we again sort them by their signatures. We keep traversing by removing nodes from the queue and perform the same process. If the parallel traversal completes successfully, the two group nodes are considered similar.

Time complexity. For the blocking step, we insert each group node to the cluster hash map based on its key, which is already included in the hierarchy. Since insertion in a hash map takes $O(1)$ time on average, the blocking step takes $O(N)$ time if the model has N group nodes. For comparing group nodes in each cluster, checking similarity for two subgraphs with V nodes and E edges takes $O(V+E)$ time. Comparing many group nodes can be expensive. However, from our experiment with sample models, dissimilar group nodes never have the same blocking key. Therefore, in practice, we only need to perform subgraph similarity just to verify that a pair of group nodes are similar.

[1] Since we detect templates in smaller group nodes first, each child of the examined group node always already has a template identifier assigned.

5.4 Overlaying Additional Quantitative Data

In addition to structural similarity, users can use color to encode other quantitative information (T5).

The *device distribution view* helps users understand how computation is distributed across multiple devices. As shown on the right, this view colors each operation according to the device it is allocated to run on. A group node is colored proportionally to the use of different devices for the operations inside it.

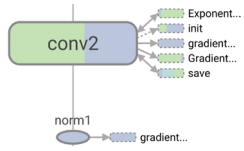


Fig. 7. Nodes in the device distribution view.

The *compute time* and *memory* views enables users to find and detect memory and computation bottleneck in their models. These views color nodes using a single-hue color ramp: nodes with higher compute time or memory usage have more saturation.

5.5 Implementation

TensorFlow Graph Visualizer is an open-source, web-based visualization. The source code is available in TensorBoard's repository [5].

We generate the layout by recursively computing (depth first) the flow layout for each expanded group's subgraph. For example, in Figure 5-c, the layout for the subgraphs inside `train` and `weights` are calculated first. The layout for the root node's subgraphs are then computed. To include embedded inputs and output, we compute the bounding box of each node, including all of its embeddings, and then calculate the layout for these bounding boxes. Next, we adjust the anchor points of the edges so that they all connect directly to the node's shape. Finally, we render the graph in SVG and animate the graph during expansion using D3.js [12].

We use Dagre, a Javascript library for a Sugiyama-style flow layout [50], to compute a layout for each subgraph. Although Dagre sometimes produces unnecessary edge crossings, the library enables us to build clustered graphs and apply our graph extraction strategy to produce layouts that are overall legible. We consider the reduction of edge crossings in each subgraph beyond the scope of this work. However, prior edge crossing reduction techniques [22, 24, 29, 33] can be directly applied to improve the layout of each subgraph.

6 Neural Network Exploration Scenarios

This section describes example scenarios for using the TensorFlow Graph Visualizer to explore neural networks in TensorFlow.

Scenario 1: Exploring a Convolutional Network

A user wants to learn about **`tf_cifar`**, an example convolutional neural network for classifying images from the CIFAR-10 benchmark dataset [36]. Since the model is based on roughly seven hundreds line of code (included in supplementary material), the user looks at the visualizer along with the code to understand the model structure. Figure 1-a shows an overview of the graph (T1). The *main graph* primarily includes input processing operations and layers of calculation networks for making inference. On the side, the *auxiliary nodes* include non-critical nodes that are extracted to declutter the layout since they connect to many layers in the model.

Consider the *main graph*, the model first reads input images. To increase the training dataset size, the model applies randomized image processing (process_image) such as adjusting brightness and contrast to produce additional data. It then shuffles the images and divides them into batches for training (shuffle_batch). The middle part of the graph shows how the model makes inferences. In this part, the model first stacks two sets of *convolution* layers [38], which efficiently use shared parameters to produce higher-level features from the input. As the convolution layers have identical nested structure, their nodes share the same color (T2). Following each of the convolution layers, the model includes a *max-pooling* operation (pool1-2) to downsample the representation size and make the model invariant to low-level transformations, as well as a *local response normalization* operation [36] (norm1-2) to reduce overfitting. The final part of the inference network contains *fully connected layers* (fully3-4), similar to layers in traditional neural networks, for performing high-level reasoning based on the produced features. After the fully connected layers, the model uses the softmax (multinomial logistic regression) module to calculate the probability distribution between different classification categories. Finally, the model computes the cross_entropy between the prediction output and the labels of the input images as its loss function.

Expanding the conv1 module (Figure 1-b) to see its nested structure (T3), the user observes that the module composes a Conv2D operation with weights and biases variables, and forwards the output to a node named conv1. Curious about the conv1 node, the user selects it to inspect details in the information card (T4). The node is a rectified linear unit (Relu), an activation function that enables convolutional networks to make breakthroughs in recognition tasks [36, 37]. From the weights, the module also computes L2Loss and passes the output to multiple auxiliary nodes including total_loss and train.

Glancing at the *auxiliary nodes*, the user sees several groups that connect to all convolution and fully connected layers (conv1-2 and fully3-4). These include groups for state saving (save), error reporting (report_uninitialized_variables), model initialization, training and total_loss function calculation. The model also contains auxiliary nodes (*e.g.,* group_deps and init) in which all edges are control dependencies to manage execution order in the dataflow.

After understanding the model structure, the user trains the model. During the training, she uses the memory and compute views to examine parts of the graphs that take a lot of memory and compute time during the training (T5). The user also uses the names of the nodes in the graph to select summary plots in TensorBoard. As the user experiments and modifies the model, she uses the visualizer to inspect modifed parts in the graph to verify her changes in the code.

Scenario 2: Exploring a Very Deep Network

Figure 8 shows an implementation of Inception [51] (**tf_inception**), a deep network architecture that had the top classification result in the ImageNet 2014. Due to space limitation, we show a version of the model that excludes training and bookkeeping operations. From the overview in Figure 8-a (T1), the user sees that the model contains about twenty top-level layers. The bottom part contains input processing operations. The topmost node is the softmax module for determining classification output.

From the overview, the user recognizes that some nodes share the same colors and thus have identical nested structure (T2). The lower part of the model contains the same convolution module (*e.g.,* conv1-4), with identical max-pooling layers occasionally interleaving in the middle. The upper part contains the *inception* mixed modules [51] that combine convolutional and pooling layers. Two groups of these mixed modules (mixed and mixed_1-2, and mixed_4-7) are identical, while other mixed modules shown in grey (mixed_3 and mixed_8-10) have unique structures.

The user can expand particular modules to explore their nested structure (T3). Figure 8-b shows the expanded state of mixed_4 and mixed_5, confirming that they share the same structure. The user observes that each of these inception modules produces high-level features using 4 parallel pathways (conv, tower, tower_1, and tower_2) that are later concatenated (join). Expanding each tower unit in Figure 8-c shows that all of these towers contain identical convolutional modules (conv). As these modules are blue, the user realizes that they are also identical to other blue convolutional modules in the lower layers (in Figure 8-a). Finally, expanding a convolutional module (Figure 8-c, left) shows individual operations that form the module.

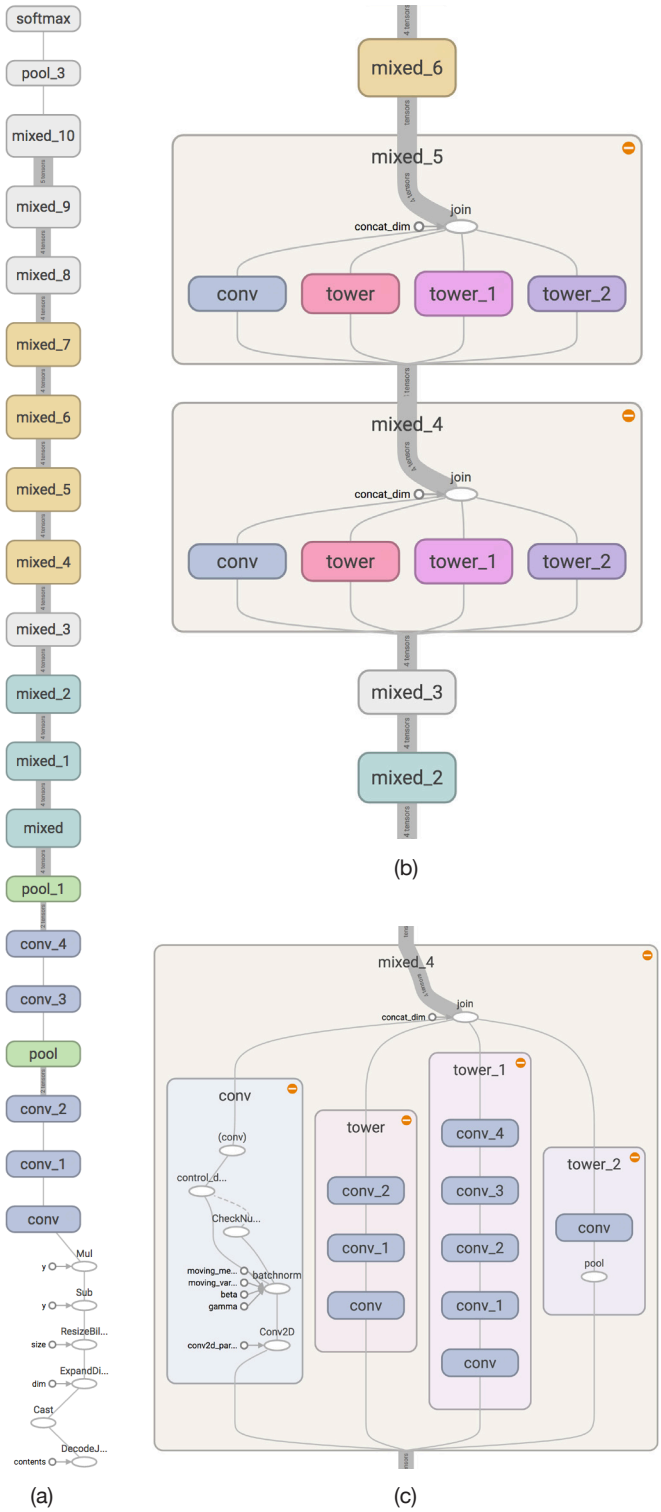


Fig. 8. Exploring an implementation of Inception (**tf_inception**) [51], a deep network architecture that won the ImageNet 2014 competiton. (a) The overview shows identical modules with the same colors. (b) Expanding two identical modules (mixed_4-5) displays similar substructures. (c) Drilling down, mixed_4 is composed of multiple conv modules identical to the conv modules in the top-level graph. Expanding the conv module on the left reveals nested operations that form the module.

7 Qualitative Feedback

We gathered feedback from real-world users of the visualizer in many ways. Within our company, we sent out a structured questionnaire for feedback; we also observed and talked directly with developers who have used the tool. In addition, since the tool has been released publicly, we also collect feedback and comments from online forums.

7.1 Internal feedback

Here we report on internal feedback that was collected from: (1) formal questionnaire and (2) observations of usage "in the wild".

Structured questionnaire

After the launch of TensorFlow, we followed up with 8 internal users of the Graph Visualizer to better understand why they used the visualization and what, if any, value they derived from it. We sent a structured questionnaire to ask them about their goals, usage, and problems.

Of our respondents, 5 were researchers (experimenting with and developing models) and 3 were engineers (applying existing models to products). None of them were the beta users that met with us weekly during the design process. Before using our visualization, three users said they had built their own tools to look at model architecture but had not been satisfied. The overall response was positive, with a good measure of the excitement resting on the interactive nature of the graph:

"It's great - visually appealing and the structure exploration seems very effective."

"This is absolutely amazing! I especially love the ability to define custom collapsible units by using the / symbol - it does wonders in cleaning up my graph. "

When asked about the types of tasks they tried to accomplish, their answers ranged from checking model architecture, to inspecting what hardware different parts of the network were running on, to debugging model structure. Here are quotes that speak to user goals:

"Understanding what my code actually produced. We had layers of functions and configs and arguments and it's good to verify we got what we intended"

"Find the name of a tensor so that I could do further exploration (like sampling that tensor over different inputs or seeing the evolution of a particular input)"

"I needed to find out which device was running what, and I did."

"Sanity-checking some specific part of the graph to make sure things are as expected"

Observed usage

Besides the questionnaire, we informally observed how the visualizer was being used "in the wild". Without our intervention, we take note of conversations in internal mailing lists and look at graphs made by users. From this examination, we discover a number of usage patterns.

Many users deliberately add namespaces to their models to ensure graph legibility. They iteratively modify namespaces until the visualization became a reasonable match to the mental model they had of their system, especially if they have to share their models with others. Our belief that users would annotate the graph to ensure visual legibility was not a foregone conclusion. These observations validate our decision to exploit user-specified namespaces to build a clustered graph. Moreover, they indicate that the visualizer is useful for users.

Many users also create screenshots of the graphs (or their parts) to communicate about deep learning systems. Sometimes this involves sharing "before and after" screenshots that show the graph as it changed during debugging. Other times the images are used simply as a visual reference to a particular piece of the system. Images of different graphs also regularly show up in the official TensorFlow tutorials and third-party articles, attesting to their value as didactic illustrations.

7.2 Public feedback

To gather public feedback outside our company, we also searched online for user reactions to the Graph Visualizer. Unlike with internal users, where demand characteristics [44] can be an issue, external users have no incentive to "be nice" about the diagrams the Graph Visualizer creates. We found that online reviews of TensorFlow have repeatedly called out the Graph Visualizer as differentiating the system from other deep learning platforms. Some typical examples:

"I think there are two main differences at the moment, comparing it to the more mainstream libraries: 1: The visualization module (TensorBoard): One of the main lacking areas of almost all open source Machine Learning packages, was the ability to visualize model and follow the computation pipeline." [Quora]

"We believe visualization is really fundamental to the creative process and our ability to develop better models. So, visualization tools like TensorBoard are a great step in the right direction." [Indico]

All of the comments we found have positive tone; we did not see users complaining about the tool. One reason may be that having any visualization at all is an improvement over the norm. The warm reception suggests that this is a design problem worth solving, and (as one commenter says) our tool is a least a "step in the right direction."

In addition to explicit feedback, we found many examples where people use screenshots of the Graph Visualizer to describe specific applications or explain intricacies of a particular type of model they built. These cases show that users find the visualizer helpful for communicating their ideas. Many users have also created tutorials that explain how to author namespaces to help the visualizer produce hierarchical clusters that matches the semantics of the model.

8 Conclusion and Future Work

Deep learning models are becoming increasingly important in many applications. Understanding and debugging these models, however, remains a major issue. This paper describes a design study of a visualization tool that tackles one aspect of this challenge: interactive exploration of the dataflow architecture behind machine learning models.

Our design process emphasizes understanding of both users and data: we describe a task analysis for developers of deep learning models, and outline the layout challenges presented by model structures. We then present a series of graph transformation to address these challenges, and demonstrate usage scenarios. Finally, we discuss user reactions, which indicate that the visualizer addresses users' need.

In the context of TensorFlow, there are many natural next steps. Some users have asked for "two-way" visualizations that allow direct editing of a graph. Direct manipulation capabilities in the graph could ease the creation and modification of machine learning models. Features for comparing multiple models could be helpful as well.

Another lesson may be applicable to other systems that visualize graphs with similar structures. The strategy of extracting non-critical nodes seems successful: viewers apparently understand the overall structure of the graph despite the lack of direct edges. While our heuristics to determine non-critical nodes are application-specific, the strategy of extracting non-critical nodes may be applicable for heterogeneous graphs in other domains.

An intriguing finding is that developers were willing to change their own code in the interest of improving the visualization, manually adding metadata to their graph in order to clarify the layout. For one thing, this behavior shows that users derived significant value from the visualizations. More importantly, it suggests that in other contexts, designers need not feel bound by the data at hand; with the right visualization, and a tight feedback loop between artifact visualization and creation, users may be willing to add critical pieces of metadata. This is a welcome sign for visualization creators.

Finally, developer reactions also suggest a heartfelt desire for better ways to understand machine learning. This is an area in which data is central, but the tools have not matured, and users often feel they operate in the dark. Visualization may have an important role to play.

Acknowledgments

We thank our colleagues at Google for advice and feedback during the design process: Greg Corrado, Jeff Dean, Matthieu Devin, Chris Olah, Koray Kavukcuoglu, Jon Shlens, Michael Terry, as well as our early users. We also thank UW Interactive Data Lab members and Supasorn Suwajanakorn for their comments on this manuscript.

REFERENCES

[1] colah's blog. http://colah.github.io/. Accessed: 2017-03-15.

[2] ConvNetJS. http://cs.stanford.edu/people/karpathy/convnetjs/. Accessed: 2017-03-15.

[3] Keras: Deep learning library for theano and tensorflow. https://keras.io/. Accessed: 2017-03-15.

[4] Labview. http://www.ni.com/labview/. Accessed: 2016-03-15.

[5] The TensorBoard repository on GitHub. http://github.com/tensorflow/tensorboard. Accessed: 2017-06-15.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] J. Abello, F. Van Ham, and N. Krishnan. ASK-Graphview: A large scale graph visualization system. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):669–676, 2006.

[8] D. Archambault, T. Munzner, and D. Auber. Grouseflocks: Steerable exploration of graph hierarchy space. *Visualization and Computer Graphics, IEEE Transactions on*, 14(4):900–913, 2008.

[9] D. Archambault, H. C. Purchase, and B. Pinaud. The readability of path-preserving clusterings of graphs. In *Computer Graphics Forum*, volume 29, pages 1173–1182. Wiley Online Library, 2010.

[10] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*, pages 133–140. IEEE, 2007.

[11] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[12] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[13] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In *Graph Drawing*, pages 31–44. Springer, 2001.

[14] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[15] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[17] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *Knowledge and Data Engineering, IEEE Transactions on*, 24(9):1537–1555, 2012.

[18] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, IDIAP, 2002.

[19] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[20] E. Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

[21] C. Dunne and B. Shneiderman. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3247–3256. ACM, 2013.

[22] T. Dwyer and Y. Koren. Dig-CoLa: directed graph layout through constrained energy minimization. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 65–72. IEEE, 2005.

[23] T. Dwyer, Y. Koren, and K. Marriott. Drawing directed graphs using quadratic programming. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):536–548, 2006.

[24] T. Dwyer, Y. Koren, and K. Marriott. IPSep-CoLa: An incremental procedure for separation constraint layout of graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):821–828, 2006.

[25] T. Dwyer, Y. Koren, and K. Marriott. Constrained graph layout by stress majorization and gradient projection. *Discrete Mathematics*, 309(7):1895–1908, 2009.

[26] T. Dwyer, K. Marriott, and M. Wybrow. Topology preserving constrained graph layout. In *International Symposium on Graph Drawing*, pages 230–241. Springer, 2008.

[27] P. Eades, Q.-W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In *Graph Drawing*, pages 113–128. Springer, 1996.

[28] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz and dynagraph—static and dynamic graph drawing tools. In *Graph Drawing Software*, pages 127–148. Springer, 2004.

[29] M. Forster. Applying crossing reduction strategies to layered compound graphs. In *Graph Drawing*, pages 276–284. Springer, 2002.

[30] E. R. Gansner, Y. Koren, and S. C. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):457–468, 2005.

[31] E. R. Gansner, E. Koutsofios, S. C. North, and G.-P. Vo. A technique for drawing directed graphs. *Software Engineering, IEEE Transactions on*, 19(3):214–230, 1993.

[32] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[33] M. Jünger and P. Mutzel. *Exact and heuristic algorithms for 2-layer straightline crossing minimization*, pages 337–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[34] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[35] S. Kieffer, T. Dwyer, K. Marriott, and M. Wybrow. Hola: Human-like orthogonal network layout. *IEEE transactions on visualization and computer graphics*, 22(1):349–358, 2016.

[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[37] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[38] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[39] 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.

[40] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, 1995.

[41] D. Moritz, D. Halperin, B. Howe, and J. Heer. Perfopticon: Visual query analysis for distributed databases. In *Computer Graphics Forum*, volume 34, pages 71–80. Wiley Online Library, 2015.

[42] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.

[43] S. C. North and G. Woodhull. Online hierarchical graph drawing. In *International Symposium on Graph Drawing*, pages 232–246. Springer, 2001.

[44] M. T. Orne. Demand characteristics and the concept of quasi-controls. *Artifacts in Behavioral Research: Robert Rosenthal and Ralph L. Rosnow's Classic Books*, page 110, 2009.

[45] G. Sander. Layout of compound directed graphs. Technical report, Saarlandische Universitat und Landesbibliothek, Postfach 151141, 66041 Saarbracken, 1996.

[46] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 669–678. ACM, 2014.

[47] D. Smilkov, S. Carter, D. Sculley, F. B. Viegas, and M. Wattenberg. Direct manipulation visualization of deep networks. In *ICML Workshop on Visualization for Deep Learning*, 2016.

[48] H. Strobelt, S. Gehrmann, B. Huber, H. Pfister, and A. M. Rush. Visual analysis of hidden state dynamics in recurrent neural networks. *arXiv preprint arXiv:1606.07461*, 2016.

[49] K. Sugiyama and K. Misue. Visualization of structural information: Au-

tomatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, 1991.

[50] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, 1981.

[51] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[52] D. Thompson, J. Braun, and R. Ford. *OpenDX: paths to visualization; materials used for learning OpenDX the open source derivative of IBM's visualization Data Explorer*. Visualization and Imagery Solutions, 2004.

[53] J. W. Tukey. Exploratory data analysis. 1977.

[54] F. Y. Tzeng and K. L. Ma. Opening the black box - data driven visualization of neural networks. In *VIS 05. IEEE Visualization, 2005.*, pages 383–390, Oct 2005.

[55] F. Van Ham and J. J. Van Wijk. Interactive visualization of small world graphs. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 199–206. IEEE, 2004.

[56] F. Van Ham and M. Wattenberg. Centrality based visualization of small world graphs. In *Computer Graphics Forum*, volume 27, pages 975–982. Wiley Online Library, 2008.

[57] V. Yoghourdjian, T. Dwyer, G. Gange, S. Kieffer, K. Klein, and K. Marriott. High-quality ultra-compact grid layout of grouped networks. *IEEE transactions on visualization and computer graphics*, 22(1):339–348, 2016.

VISUALIZING DATAFLOW GRAPHS OF DEEP LEARNING MODELS IN TENSORFLOW: SUPPLEMENTARY MATERIAL

A TENSORBOARD

TensorBoard is Tensorflow's dashboard tool, which allows users to visualize their TensorFlow models, plot quantitative metrics about their execution, and show additional data like images that pass through the models. The TensorFlow Graph Visualizer is included as the *Graph View* in TensorBoard. Besides the Graph Visualizer, TensorBoard provides others views (Figure S-1) for inspecting scalar, histogram, images, and audio data. To log data in the models, users can add summary operations that takes operation they desire to log as inputs. To visualize the graphs, user can include dataflow information for the Graph Visualizer in the log data. Users can also optionally include additional metadata about runtime statistics including total memory usage, total compute time, and tensor shapes.

For specific information about how to setup and run TensorBoard, please refer to TensorFlow tutorials[1].

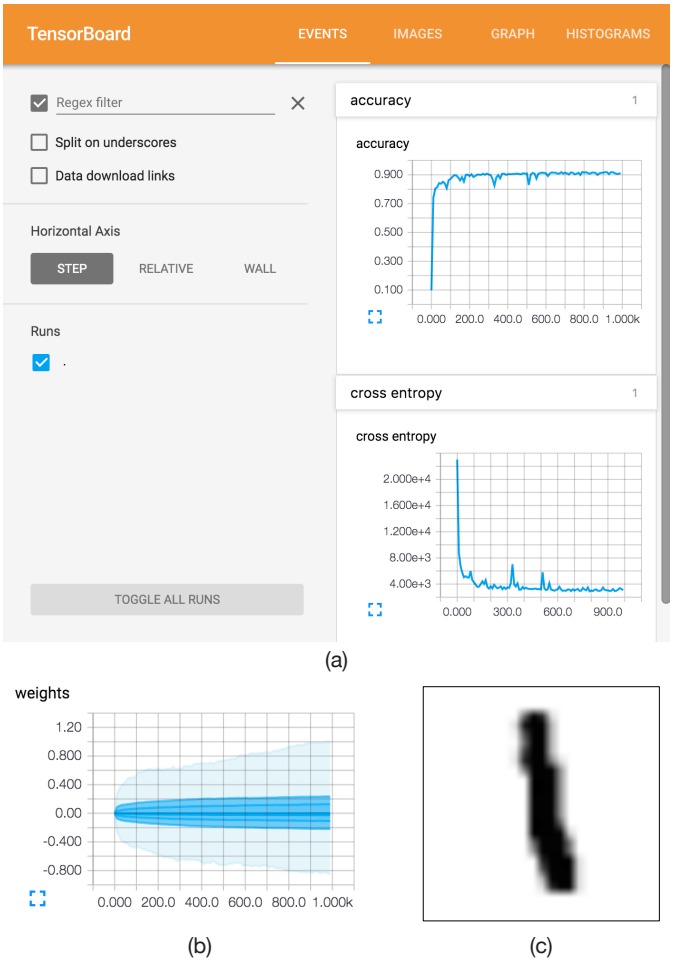


Fig. S-1. Summary log viewers in TensorBoard, TensorFlow's dashboard. (a) The *events view* showing plots of accuracy over time for the **tf_mnist_simple** model. (b) The *histogram view* showing distribution of the `weights` variable's tensor values over time. (c) The *image view* showing a handwritten digit image input.

B EXAMPLE CODE

tf_mnist_simple

The Listing 1 below shows an example Python code snippet that declares a simple model for classifying handwritten digits, derived from TensorFlow's "MNIST For ML Beginners" getting started tutorial[2].

The code first defines a placeholder for the input and initializes variables for model parameters (Lines 1-5). It then calculates $y = softmax(Wx+b)$ (Lines 6-7), which produces a probability distribution for each of the ten digits (0-9), and trains the model by minimizing cross entropy using a gradient descent optimizer (Lines 11-17). Finally, it evaluates the model by comparing the digit with highest probability for each data point with the correct label and calculating the ratio between correct predictions and the number of all data points (Lines 18-22). Namespaces are provided for operations via the `name` parameter of operation constructor methods (Lines 2-5), and the `tf.name_scope` method (Lines 6, 12, 15, and 19), which wraps operations with a common namespace. Certain operations are also logged with summary operations for debugging (Lines 8-10, 14, 22).

```
1  # Input placeholder, params variable, summary log
2  x = tf.placeholder(tf.float32, [None, 784],
       name='x-input')
3  y_ = tf.placeholder(tf.float32, [None, 10],
       name='y-input')
4  W = tf.Variable(tf.zeros([784, 10]), name='weights')
5  b = tf.Variable(tf.zeros([10]), name='bias')
6  with tf.name_scope('Wx_b'):
7    y = tf.nn.softmax(tf.matmul(x, W) + b)
8  tf.histogram_summary('weights', W)
9  tf.histogram_summary('biases', b)
10 tf.histogram_summary('y', y)
11 # Define loss and optimizer
12 with tf.name_scope('xent'):
13   cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
14   tf.scalar_summary('cross entropy', cross_entropy)
15 with tf.name_scope('train'):
16   train_step = tf.train.GradientDescentOptimizer(
17        FLAGS.learning_rate).minimize(cross_entropy)
18 # Compare predicted y with input y_
19 with tf.name_scope('test'):
20   correct_prediction = tf.equal(tf.argmax(y, 1),
       tf.argmax(y_, 1))
21   accuracy = tf.reduce_mean(tf.cast(correct_prediction,
       tf.float32))
22   tf.scalar_summary('accuracy', accuracy)
23 # Write log
24 merged = tf.merge_all_summaries()
25 writer = tf.train.SummaryWriter(FLAGS.summaries_dir,
26   sess.graph.as_graph_def(add_shapes=True))
```

Listing 1. A example Python snippet that declares a simple model for classifying handwritten digits in the MNIST dataset with softmax regression (**tf_mnist_simple**).

tf_cifar

The source code is in `cifar10.zip` in the supplementary material zip file. See `cifar10_train.py` for the training code.

tf_inception

The code can be found online in the `inception` directory in the `tensorflow/models` repository on GitHub[3].

[1] https://www.tensorflow.org/get_started/summaries_and_tensorboard

[2] https://www.tensorflow.org/get_started/mnist/beginners

[3] https://github.com/tensorflow/models