INTENT: Interactive Tensor Transformation Synthesis

Zhanhui Zhou*
University of Michigan
Ann Arbor, MI, USA
zhanhui@umich.edu

Man To Tang*
Purdue University
West Lafayette, IN, USA
tang426@purdue.edu

Qiping Pan*
University of Michigan
Ann Arbor, MI, USA
panqp@umich.edu

Shangyin Tan
Purdue University
West Lafayette, IN, USA
tan279@purdue.edu

Xinyu Wang
University of Michigan
Ann Arbor, MI, USA
xwangsd@umich.edu

Tianyi Zhang
Purdue University
West Lafayette, IN, USA
tianyi@purdue.edu

ABSTRACT

There is a growing interest in adopting Deep Learning (DL) given its superior performance in many domains. However, modern DL frameworks such as TensorFlow often come with a steep learning curve. In this work, we propose INTENT, an interactive system that infers user intent and generates corresponding TensorFlow code on behalf of users. INTENT helps users understand and validate the semantics of generated code by rendering individual tensor transformation steps with intermediate results and element-wise data provenance. Users can further guide INTENT by marking certain TensorFlow operators as desired or undesired, or directly manipulating the generated code. A within-subjects user study with 18 participants shows that users can finish programming tasks in TensorFlow more successfully with only half the time, compared with a variant of INTENT that has no interaction or visualization support.

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; **Interactive systems and tools**.

KEYWORDS

Program Synthesis, Deep Learning, Interactive Visualization

ACM Reference Format:
Zhanhui Zhou, Man To Tang, Qiping Pan, Shangyin Tan, Xinyu Wang, and Tianyi Zhang. 2022. INTENT: Interactive Tensor Transformation Synthesis. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3526113.3545653

1 INTRODUCTION

The use of Deep Learning (DL) has grown rapidly in the past decade. Due to its superior performance, DL has found its application in various domains and continued to fascinate us with new applications such as cloud migration [6]. Thanks to the development of modern DL frameworks such as TensorFlow, DL has become more accessible to regular programmers [12]. However, there is still a steep learning curve for regular programmers to adopt DL in their own practices. A recent survey found that programmers encountered a variety of hurdles when learning machine learning, e.g., lack of conceptual and mathematical understanding, cryptic API or syntax design, etc. [3].

Indeed, DL programming is quite different from traditional programming. Unlike traditional programming, tensors (i.e., multi-dimensional arrays) are the first-class citizen in DL computation, including not only internal model computation but also data preprocessing and model performance calculation. For instance, a programmer needs to use special tensor operators such as `tf.where` and `tf.cond` in TensorFlow to perform conditional operations on a tensor, rather than using traditional `if-else` statements. While tensors are a natural and efficient representation of massive, high-dimensional data in DL, writing code to operate on tensors requires solid linear algebra and calculus knowledge. Furthermore, modern DL frameworks define a large number of tensor transformation operators (e.g., about 500 in TensorFlow) but suffer from insufficient documentation and poor API design [3, 53, 55]. For example, some tensor operators have cryptic names that do not match the underlying computation, e.g., `tf.eye` for constructing an identity matrix. Some operators have similar names but different functionalities, such as `tf.argmax` vs. `tf.reduce_max` and `tf.ones` vs. `tf.ones_like`. These issues lead to significant design barriers and selection barriers [18] in DL programming.

In this work, we propose INTENT, an interactive program synthesis system that generates tensor transformation programs in TensorFlow on behalf of users. Users can specify a desired transformation in natural language and supplement input-output examples to illustrate the transformation. INTENT extends an existing synthesis algorithm from TF-Coder [44] to handle such multi-modal user specifications. Specifically, it infers the weights of tensor operators using a combination of an NLP model and a multi-layer perceptron. Then, it performs a bottom-up enumerative search to synthesize a tensor transformation program that satisfies the given examples. Based on previous synthesis results, users can further mark desired or undesired operators to adjust their weights and guide the synthesizer towards promising synthesis directions.

One core usability challenge addressed by INTENT is the challenge of understanding and validating synthesized code, as identified by prior work [11, 22, 37]. This challenge is exacerbated in

*The first three authors contributed equally to this work.

UIST '22, October 29-November 2, 2022, Bend, OR, USA

ACM ISBN 978-1-4503-9320-1/22/10.
https://doi.org/10.1145/3526113.3545653

DL programming due to cryptic API names and sophisticated linear algebraic operations on tensors. To address this challenge, we designed a novel interactive visualization that renders the intermediate steps of a tensor transformation program in a dataflow diagram. The dataflow diagram is further augmented with element-wise data provenance, through which users can trace the flow of specific elements inside a tensor.

Sometimes, a synthesizer may have generated a solution close to the final solution. However, due to the exponentially-grown search space, users often find it hard to guide the synthesizer to generate the final solution from this point. This is dubbed *the last mileage problem* in program synthesis by prior work [52]. In such cases, users prefer to directly edit the synthesized code instead. INTENT supports this need with an in situ program editing and validation feature, so users do not have to switch to an external Python environment to edit and test the code.

To evaluate the usefulness of INTENT, we conducted a within-subjects user study with 18 programmers with different levels of ML experience. We created a comparison baseline of INTENT by disabling the interactive visualization and validation support. The result shows that participants using INTENT finished the assigned TensorFlow coding tasks more successfully with only half of the time on average. Specifically, when using the baseline tool, 8 participants switched to a Jupyter notebook on Google Colab and tested some variations of synthesized code, while no participants did that when using INTENT. In the post-study survey, participants felt more confident about the synthesis results—6.72 vs. 5.44 on average in a 7-point Likert scale confidence rating question (t-test: p=0.005). These results imply that augmenting intelligent code generation systems with interactive visualization and validation support in tightly-coordinated views can significantly improve programmer productivity when writing TensorFlow code.

Overall, we make the following contributions:

- A tensor transformation synthesis system with rich interaction support for non-expert DL programmers. We have open-sourced our system on GitHub.[1]
- A novel dataflow visualization of tensor transformation code with element-wise data provenance to help users understand tensor transformation code.
- A within-subjects study with 18 programmers to demonstrate that INTENT improves both the programming productivity and programmers' confidence on program synthesis.

2 RELATED WORK

2.1 Programming-by-Example Systems

Programming-by-Example (PBE), also termed as Programming-by-Demonstration (PBD), is a technique that automatically generates programs from user-provided examples. There is a long history of developing PBE systems to support non-experts in writing programs [20, 27]. In 1986, Myers and Buxton developed one of the first PBE systems called Pedriot, which allowed users to create UI widgets such as menus and scrollbars by demonstration [35]. Various PBE systems have been proposed since then [4, 5, 7, 9, 10, 17, 21, 23–26, 28, 31–34, 36, 43, 48, 49, 51, 52, 54]. In this work, we investigate tensor transformation synthesis, which is a new but increasingly important synthesis problem. Nowadays, a growing number of people want to adopt deep learning in their own practices. However, authoring programs using deep learning frameworks such as TensorFlow is not easy. To the best of our knowledge, the only work tackling this problem is TF-Coder [44]. TF-Coder is designed as a fully automated tool, while INTENT brings new interactive support to address known usability issues in program synthesis. First, INTENT provides a novel dataflow visualization with intermediate values and element-wise provenance to help users understand synthesized TensorFlow code. By contrast, TF-Coder renders the raw code. Second, INTENT allows users to directly modify synthesized code and test it with new examples within the interface, so users do not have to switch to an external IDE and set up a testing environment. Finally, INTENT supports richer specification modality and fine-grained control of the synthesis process by specifying TensorFlow operators as desired or undesired.

2.2 Interaction Support for PBE Systems

Existing PBE systems adopt different kinds of interaction design to address common usability challenges in PBE, such as program comprehension difficulty, lack of confidence, and ambiguous user specifications. The most related interaction support to our work is communicating synthesized code to users in a more comprehensible and user-friendly way. For instance, Wrex [7] converts a synthesized program, which is initially represented in an arcane domain-specific language, to concise, readable Python code that data scientists are familiar with. FlashProg [31] translates synthesized data extraction programs to English-like descriptions. Pursuit [34] employs a comic strip metaphor to visualize data object changes (e.g., file copy) in synthesized shell scripts. Rousillon [4] visualizes synthesized web scraping programs in a block-based visual programming language called Scratch [42]. Compared with prior work, INTENT adopts dataflow visualization to render intermediate steps in a tensor transformation program. Since tensor transformation involves high-dimensional arrays and sophisticated linear algebra computation, INTENT further augments traditional dataflow visualization with element-wise data provenance to render how individual elements in a tensor are computed.

Another related interaction support is multi-modal specification. Since examples are inherently incomplete, modern PBE systems often support additional specification modalities, such as natural language descriptions [5, 16, 30, 40, 41] and voice commands [24–26], to elicit more complete specifications and reduce ambiguity. For example, Jigsaw [16] synthesizes Python code using the Pandas APIs from natural language descriptions and input-output examples. Unlike INTENT, Jigsaw focuses on automatically detecting and fixing incorrect code generated by a language model.

The feature that allows users to mark certain TensorFlow operators as desired or undesired in INTENT is inspired by Peleg et al. [39]. In [39], Peleg et al. proposed an interaction model that allows users to specify which parts of a synthesized program must be included or excluded in the next synthesis iteration and demonstrated its effectiveness in several domains. This feature also resembles the touch modality in APPINITE [25] and PUMICE [26], in which users

[1] https://github.com/ZHZisZZ/INTENT

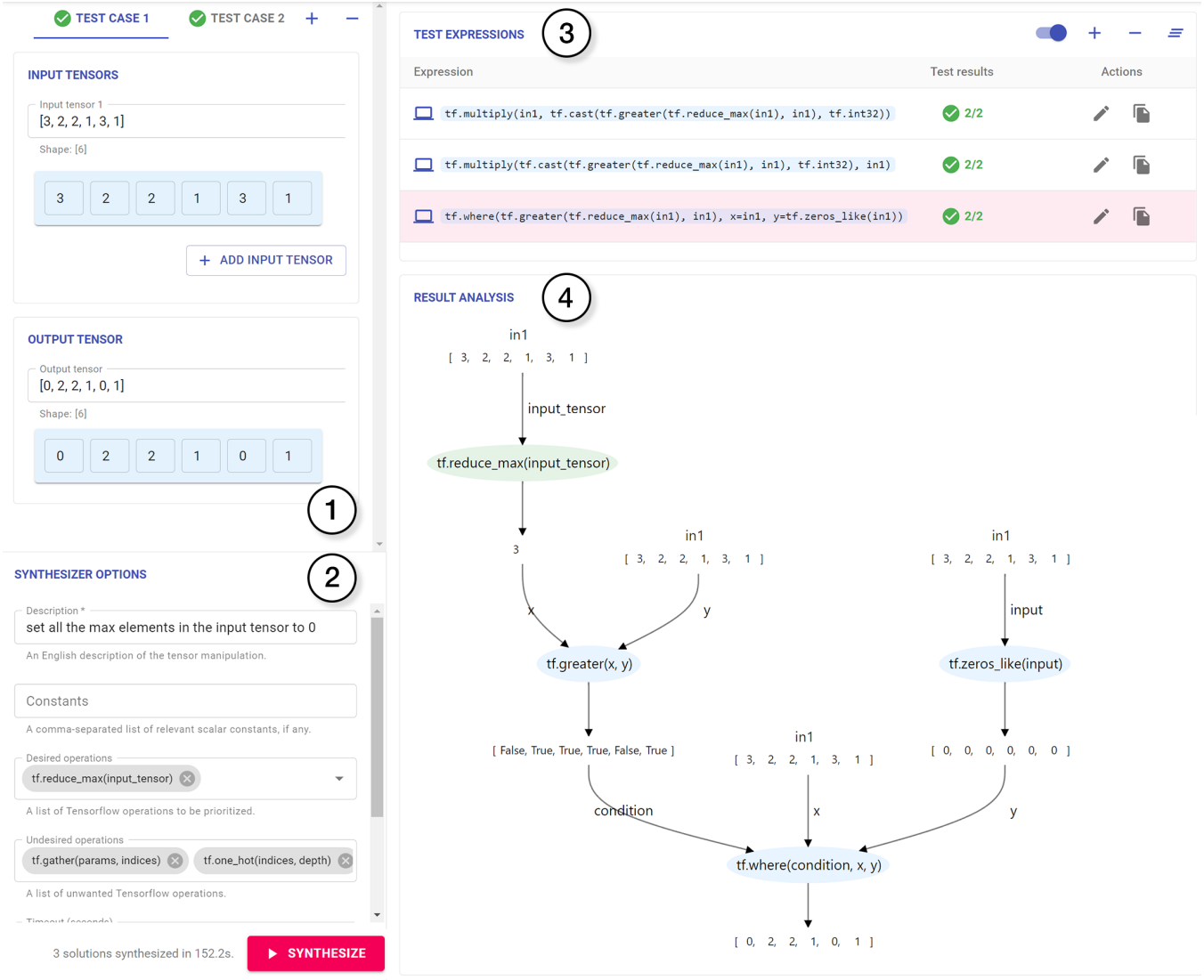

Figure 1: After adding two input-output examples (①) and a natural language description (②), INTENT automatically generates three TensorFlow programs that match the given examples (③). The dataflow diagram (④) renders a step-by-step transformation with intermediate results on the given tensor.

can touch an area in a mobile screen to pinpoint which UI widget the synthesizer should focus on.

3 USAGE SCENARIO

Suppose Kyle is a ML enthusiast who just started using TensorFlow. As part of a data preprocessing step, Kyle needs to set the maximum value in each tensor to zero. Kyle knows how to reset maximum values in a tensor using a loop in Python. However, since his tensors are very large, iterating dimensions in a tensor is not efficient. Kyle wants to use TensorFlow APIs to perform the transformation since those vectorized APIs are highly optimized, especially for GPU parallelization. Kyle searches online and finds a TensorFlow API called `tf.while_loop`, which seems to fit his need. After careful examination, Kyle finds it hard to use this API, since it requires writing two lambda expressions to control the loop, and Kyle has never written any lambda expressions before. Finally, Kyle decides to try INTENT and see if it can help him generate the code he wants.

Kyle first adds a natural language description of the desired transformation in INTENT, as shown in Figure 1 ②. He also adds a concrete example to illustrate this transformation (Figure 1 ①). After Kyle clicks the Synthesize button, the synthesizer quickly returns three programs that satisfy the given example (Figure 2). Some API calls in these programs are easy to understand, such as `tf.subtract` and `tf.multiply`. However, Kyle cannot easily tell the meaning of other API calls, such as `tf.gather` and `tf.one_hot`.

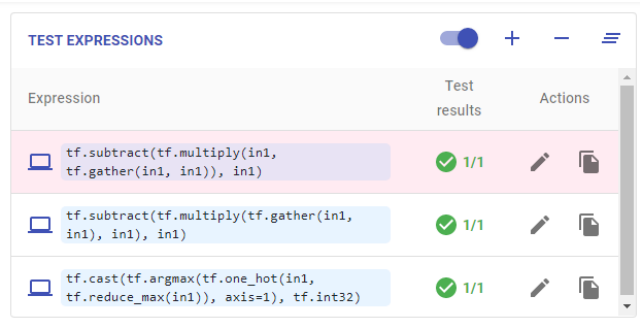

Figure 2: The synthesizer returns three TensorFlow transformation programs in the first iteration.

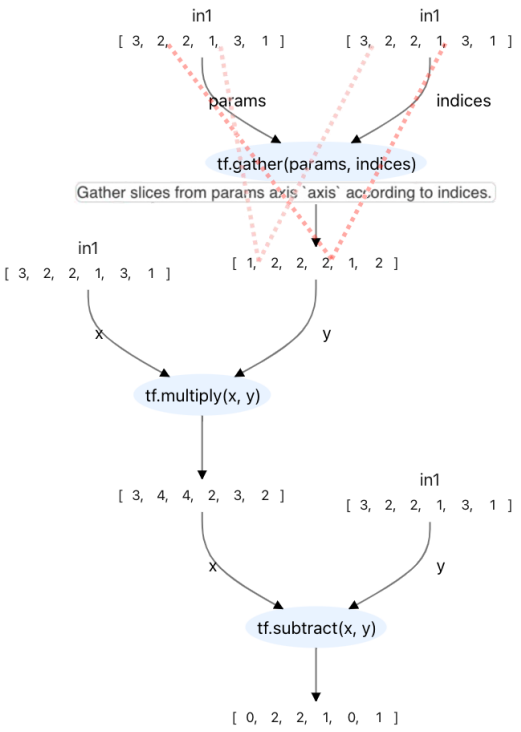

Figure 3: The dataflow diagram of the first program in Figure 2 with on-demand data provenance

In the meantime, Kyle notices that a dataflow diagram (Figure 3) is rendered for the first program. Kyle also notices that `tf.gather` is computed first in the dataflow. This operation uses the input tensor in his example as the value for both the arguments of `tf.gather`. When he hovers the mouse over the `tf.gather` node, a tooltip appears to show a description of `tf.gather`. Based on the tooltip alone, Kyle makes a guess that the output of `tf.gather` is computed by gathering elements from the `params` argument based on the elements in the `indices` argument. Nevertheless, he is still unclear how exactly this is done, e.g., which element moves to where.

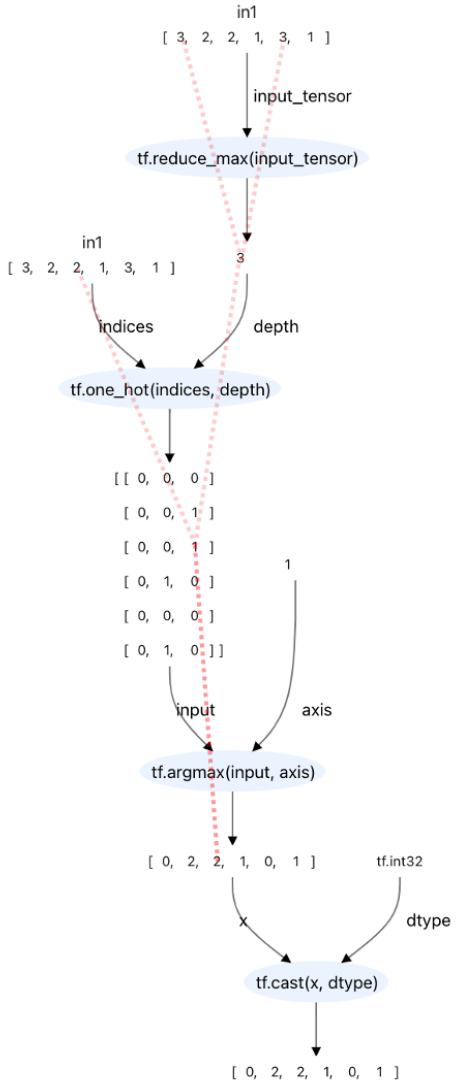

Figure 4: The dataflow diagram of the third program in Figure 2 with on-demand data provenance

Kyle clicks on the first element in the output tensor of `tf.gather`. Then two red dotted lines appear, pointing to the 4th element in the `params` argument and the 1st element in the `indices` argument. Kyle also notices that the 4th element in `params` has the same value as the 1st element in the output tensor, and the 1st element in `indices` happens to be the index of the 4th element in `params`. Kyle then clicks on the 4th element in the output tensor. The two red dotted lines show that this element has the same value as the 2nd element in the `params` argument and the connected element in `indices` is the index of the 2nd element in `params`. Now, Kype confirms that the output tensor consists of the elements in `params` but these elements are reordered based on the elements in `indices`.

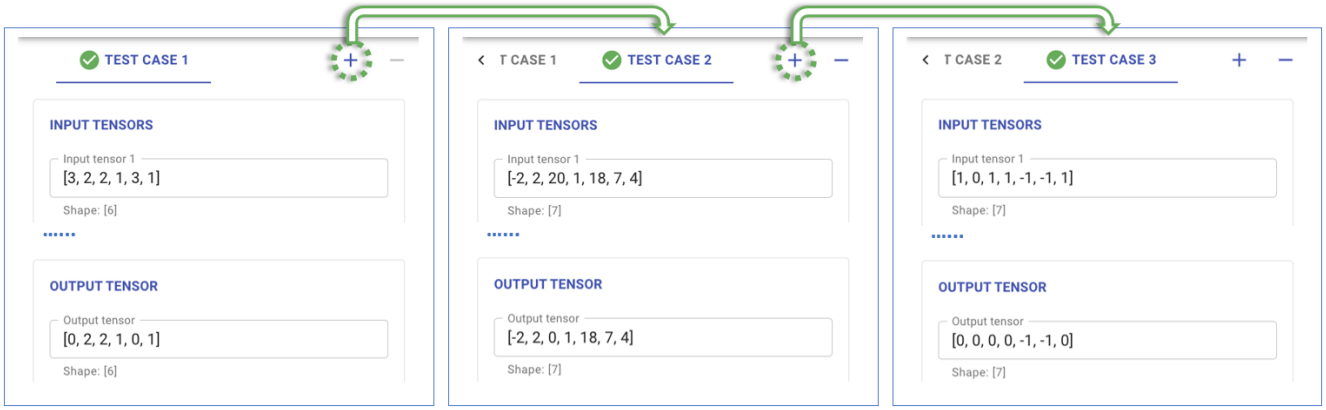

Figure 5: Kyle provides two additional test cases to verify the current programs are correct by clicking the plus icon.

Figure 6: After Kyle modifies the last program, INTENT **automatically evaluates it on the given examples and reveals that this edit leads to failures on all three examples.**

Kyle then looks at the following two operations, `tf.multiply` and `tf.subtract`, in the dataflow diagram. He realizes that this program just happens to transform the input tensor to the expected output but has nothing to do with setting the max value in a tensor to zero. The second program is the same as the first one with the arguments of `tf.multiply` swapped. Kyle moves on to investigate the last program in Figure 2. This program looks correct at first sight, since it uses `tf.arg_max` and `tf.reduce_max`, both of which look related to computing the max value of a given tensor. But Kyle is not sure what `tf.one_hot` does. Kyle turns to its dataflow diagram (Figure 4) and finds that `tf.one_hot` converts the input tensor into a matrix that has the depth of the max value computed by `tf.reduce_max`. This looks wrong to Kyle.

Although the third program is incorrect, Kyle finds the operation, `tf.reduce_max`, quite promising. To guide the synthesizer towards this direction, Kyle marks `tf.reduce_max` as a desired operation (Figure 1 ②). He also marks `tf.gather` and `tf.one_hot` as undesired, since these operations seem irrelevant to his goal. Then he clicks the Synthesize button again.

Given the feedback from Kyle, the synthesizer generates another set of three programs (Figure 1 ③) in the second iteration. Kyle immediately noticed the last program, which starts with a `tf.where` operation followed by `tf.greater` and `tf.reduce_max`. These three operations together look quite promising to him. Kyle looks at its the dataflow diagram (Figure 1 ③) to confirm its behavior. Kyle finds that this program first computes the max value of the input tensor and then compares the max value with the input tensor using `tf.greater`. Then it sets the max values to `False` and other values to `True`. These boolean values are then used as the condition in `tf.where` to select values from the original input tensor and a tensor with all zeros. In this way, all the max elements are replaced with 0.

Kyle wants to double-check this program on some other examples. Instead of setting up a new Python environment and configuring TensorFlow to test the generated program, Kyle uses the quick validation feature provided by INTENT. He adds two additional test cases (Figure 5). Then INTENT automatically runs the three generated programs on the new test cases. He also tweaks the last program by changing `tf.greater` to `tf.equal` but finds that the modified program fails on all three examples (Figure 6). Thus, Kyle is convinced that the original programs generated is correct. This interactive experience helps Kyle understand how the generated program works internally and gives him confidence on its behavior.

4 TOOL DESIGN AND IMPLEMENTATION

Figure 7 gives an overview of the system architecture. INTENT contains three major components—(1) a multi-modal synthesizer for tensor transformation (Section 4.1), (2) an interactive dataflow visualization that facilitates program comprehension (Section 4.2), and (3) an in situ program validation feature that allows users to quickly validate the correctness of a program (Section 4.3).

4.1 Multi-Modal Tensor Transformation Synthesis

INTENT extends the bottom-up enumerative synthesis algorithm in TF-Coder [44] to support three types of specifications:

- *Input-output examples.* Users can specify input-output tensors to demonstrate the desired transformation in the specification panel (Figure 1 ②). Different from TF-Coder[44], which only accepts one example, INTENT allows *multiple examples*. Since different examples may specify different aspects of the intended

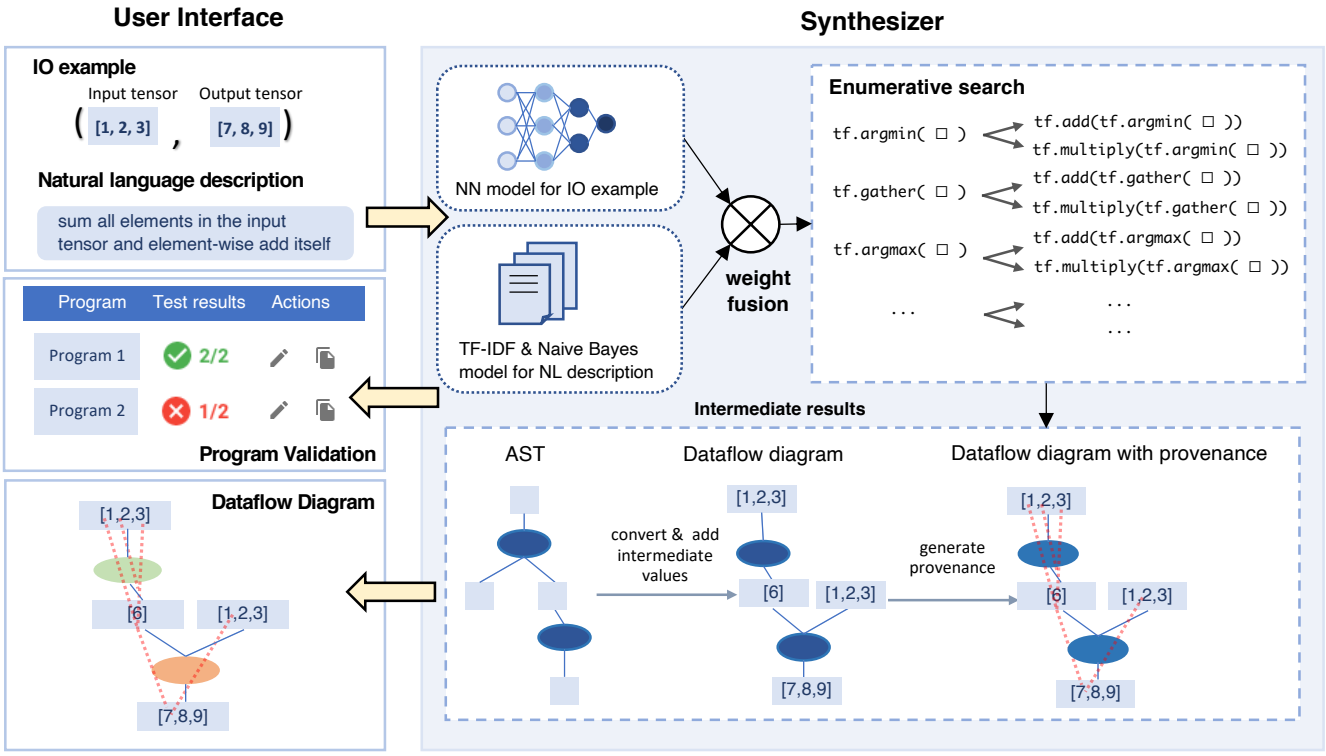

Figure 7: System Architecture

program behavior, we designed a ranking mechanism to balance multiple examples (detailed in Appendix A).

- *Natural-language descriptions.* Input-output examples are known as partial specifications, since they only express specific behaviors on specific inputs. Therefore, natural language is often used to allow users to describe the overall behavior of the desired program. INTENT supports natural language descriptions as a complementary specification modality to input-output examples; this is inherited from TF-Coder [44].
- *Desired or undesired operators.* Sometimes, users may roughly know which operators to use but do not know precisely how to use them together. Furthermore, when a synthesizer generates a wrong program, it is helpful if the synthesizer can be informed which operators are incorrect and should not be considered again in following synthesis iterations. Such hints have been shown effective in guiding the synthesis process [39]. Our interface allows users to either directly type in desired or undesired operators in the specification panel (Figure 1 ②), or right-click on an operator node in the dataflow diagram and mark it as desired or undesired. This new modality is not supported by TF-Coder [44].

Appendix A describes the multi-modal synthesis algorithm in detail. Here we briefly summarize how it works. Similar to TF-Coder [44], INTENT uses a multilayer perceptron (MLP) model to rank tensor operators in TensorFlow based on input-output examples and a naive Bayes model to rank tensor operators based on natural language descriptions. These two rankings are combined to compute the cost of each operator. The higher a tensor operator ranks, the lower its cost is. If a user provides desired or undesired operators, INTENT will adjust the cost of these operators accordingly. Given these weighted operators, INTENT constructs tensor transformation programs in a *bottom-up* manner. That is, it enumerates smaller programs first and then uses these programs as "building blocks" to construct bigger ones. The program construction process prioritizes programs with small costs (i.e., programs with highly ranked operators and few operators). This process continues until the first K programs that satisfy all given examples are found, where K is specified by the user.

4.2 Interactive Dataflow Visualization with Element-wise Data Provenance

Previous studies show that a major barrier to adopting program synthesis in practice is the difficulty of understanding and validating synthesized programs [11, 38, 54]. This challenge is exacerbated in the domain of tensor transformation since tensor transformation often involves sophisticated linear algebra computations, and TensorFlow APIs are known to be hard to comprehend. To help users understand synthesized tensor transformation programs, INTENT visualizes a program as a *dataflow diagram*, which visualizes how the input tensors are transformed by each operator eventually to the output. Since all intermediate tensors are shown step by step, the gap between the input and output tensors is greatly alleviated (Figure 3). Furthermore, as tensors are essentially multi-dimensional

arrays, users might wonder how a specific element in a tensor is computed. To support this need, we designed *element-wise data provenance* (the red dotted lines in Figure 3). Users can click an element of interest in a tensor and trace which elements in other tensors are used to compute it.

To construct the dataflow diagram of a program, INTENT first parses it to an Abstract Syntax Tree (AST) and traverses the AST to identify each tensor operator in the program. INTENT performs postorder traversal since the tensor operators executed first appear deeper in the AST than those executed later. The AST node of a tensor operator is then converted to a node in the dataflow diagram. INTENT computes data provenance in two ways. First, INTENT automatically generates the data provenance of 59 differentiable operators using the automatic differentiation of TensorFlow. Given a target element in an output tensor, INTENT computes the gradient of each element in the input tensor. If the gradient of an input element is not zero, it means the output element depends on it and therefore a data provenance can be derived. Second, we manually write specifications to compute the data provenance of the remaining 52 operators. A formal description of the data provenance computation method is provided in Appendix B.

4.3 In Situ Program Editing and Validation

As observed in prior work [52], users often get inspired by partially correct programs generated by a synthesizer. In such cases, instead of providing feedback to a synthesizer and waiting for it to regenerate the program, users prefer to directly edit the synthesized code and test its correctness. For example, suppose a user wants to write a program that *sets all maximum values in the input tensor to zeros* and she provides an input-output example below.

`in:[3, 1, 4, 5, 1, 8]` → `out:[3, 1, 4, 5, 1, 0]`

INTENT will generate a program that is very close to the correct program, as shown below.

```
tf.multiply(in, tf.cast(tf.greater( tf.constant(8) , in),
                        tf.int32)).
```

This solution is wrong since it contains a hardcoded value of 8, which happens to be the maximum value in the given input tensor. Some users, especially those experienced with TensorFlow, may spot this mistake and come up with a quick fix that replaces `tf.constant(8)` with `tf.reduce_max(in)`.

```
tf.multiply(in, tf.cast(tf.greater( tf.reduce_max(in) , in),
                        tf.int32))
```

Then, the user could validate the correctness of this modified program by running it against the input-output examples or some new test cases. Without INTENT, they have to open a Python IDE or a Jupyter notebook with TensorFlow pre-installed and then copy and paste the modified program as well as test cases to run it. INTENT eliminates this hassle by incorporating a partial program validation component. This component automatically generates a complete Python program with test stub to run a modified program with user-provided inputs from the specification panel and compares the output with user-provided outputs.

5 USER STUDY

To evaluate the usefulness and usability of INTENT, we conducted a within-subjects user study with 18 programmers with different levels of machine learning expertise. Given that INTENT is built upon TF-Coder [44], we choose TF-Coder as a comparison baseline. Since TF-Coder does not have an interface, we adapted the user interface of INTENT by disabling the new features we proposed to enable a fair comparison. Specifically, we disabled the dataflow visualization, element-wise data provenance, in situ program editing and validation, and the user annotation feature.

5.1 Participants

We recruited 18 CS graduate students (1 female, 17 male) at Purdue University using the department graduate mailing list. Most participants had adequate training in programming. 8 had more than 5 years of programming experience, 9 had 2 to 5 years, 1 had only 1 year. However, they had diverse experience with TensorFlow. 4 participants said that they were familiar with TensorFlow and had used it many times. 14 participants said they knew TensorFlow but only used it a few times. As a compensation for their participation, each participant received a $25 Amazon gift card.

5.2 Tasks

To reflect how INTENT is used in real-world programming scenarios, we selected tensor transformation tasks from the TF-Coder benchmark [44]. Since many tasks in the TF-Coder benchmark were sampled from Stack Overflow, we eliminated those tasks that users can easily find via Google Search. This was to prevent users from accidentally finding a correct solution online since we allowed participants to search online during the study. Eventually, we selected three tasks with different levels of difficulty.

The first task is considered easy to solve; the transformation is reasonably intuitive, and both synthesizers can solve it quickly in one iteration. Compared with the first task, the other two are considered harder: both synthesizers often give some plausible solutions that cannot handle some corner cases in the first iteration, thus requiring users to provide counterexamples to disambiguate their intent. The third task is considered the hardest since it involves a rarely used TensorFlow operator, `tf.boolean_mask`. When users see this solution, many may immediately consider it incorrect since it uses an operator that seems irrelevant to the task.

For each task, we created a task description in natural language and also added an input-output example to illustrate the task. During the study, we encouraged participants to use their own language and examples as input for the assigned synthesizer. Note that there exist multiple correct solutions for each task. We considered a participant to successfully complete the task once they reached any of the correct solutions. The natural language descriptions and input-output examples are listed below.

Task 1. Divide the first tensor (i.e., input1) by the second tensor (i.e., input2), but when dividing by 0, return the numerator. [Post 53643339] [2]

```
input1=[3.0, 1.0, 4.0, 5.0, 2.0, 8.0, -6.0, -7.0]
```

[2]This post has been deleted from Stack Overflow. We find a snapshot of it on Web Archive and provide the link to the snapshot instead.

```
input2=[0.5, 0.0, -2.0, 0.0, 1.0, -1.0, 0.0, 2.0]
output=[6.0, 1.0, -2.0, 5.0, 2.0, -8.0, -6.0, -3.5]

solution=tf.where(tf.cast(in2, tf.bool), x=
                    tf.divide(in1, in2), y=in1)
```

Task 2. Use the first input tensor (i.e., input1) as the boolean condition for multiplying by -10 with the second tensor. For example, if the first element in input1 is 1, then the first element in input2 will not be multiplied by -10. On the other hand, if the first element of input1 is 0, then the first element in input2 will be multiplied by -10. [From an internal Google forum]

```
input1=[1, 0, 0, 1, 0]
input2=[1, 2, 3, 4, 5]
output=[1, -20, -30, 4, -50]

solution=tf.multiply(in2, tf.gather(
                     tf.constant((-10, 1)), in1))
```

Task 3. Select the values in the second tensor where the first tensor is greater than 1. [Post 39045797]

```
input1=[2, 1, 0, 1]
input2=[3, 1, 2 ,1]
output=[3]

solution=tf.boolean_mask(in2, tf.greater(in1,
                                tf.constant(1)))
```

5.3 Protocols

Each user study starts with an introduction and consent collection. Participants were assigned two tensor transformation tasks, one to be completed with INTENT and the other to be completed with TF-Coder. To mitigate the learning effect, both task assignment order and tool assignment order were counterbalanced across participants. In total, 6 participants experienced each task in each condition. Before starting each task, participants first watched a tutorial video of the assigned synthesizer and spent about 5 minutes getting familiar with the assigned synthesizer. Then, they were given 20 minutes to finish the assigned task. A task was considered failed if participants did not find any solution after 20 minutes or if they provided a wrong solution. To simulate the real-world programming workflow, participants were allowed to search online and refer to any online resources during the study. After completing each task, participants filled out a post-task survey to give feedback. The post-task survey asked users what they liked or disliked about the assigned tool and what they wished to have. The survey also included a set of Likert-scale questions to ask users to rate the usefulness of key features in each assigned tool. To evaluate the cognitive load of finishing a task with the assigned synthesizer, we included five NASA Task Load Index questions [13] as part of the post-task survey. After all tasks were completed, participants filled out a final survey, where they directly compared the two synthesizers. We recorded each user study with the permission of the participants. A study took 64 minutes on average.

6 RESULTS

6.1 User Performance

Table 1 shows participants' performance on the three TensorFlow coding tasks using INTENT vs. TF-Coder. When using INTENT, all 18 participants successfully completed the assigned tasks within the given time. By contrast, 14 of 18 participants successfully completed the assigned tasks when using TF-Coder. Specifically, when using TF-Coder, 1 participant provided a wrong program as their final solution, and 3 participants did not finish the task within 20 minutes. Furthermore, when using INTENT, participants spent only half the time finishing the assigned task compared with using TF-Coder. The average task completion time using INTENT is 5.8 minutes, while the average task completion time using TF-Coder is 11.0 minutes. The mean difference of 6.2 minutes is statistically significant (unpaired t-test: t=4.11, df=25, p=0.0003).

We analyzed the post-task survey responses and the recordings to understand why participants using INTENT performed better. First, we found that the dataflow visualization significantly sped up the program comprehension process. Based on the recordings, all 18 participants using INTENT made heavy use of the dataflow visualization to understand the synthesized TensorFlow code. By contrast, participants using TF-Coder spent a lot of time reading official TensorFlow documentation to understand the code. Overall, 14 out of 18 participants referred to online learning resources when using TF-Coder while only 3 participants did so when using INTENT. In the post-task survey, 15 of them strongly agreed that dataflow visualization helped them understand the synthesized code (Figure 10). P17 said, "*I can see the explanation and the dataflow of the generated code, so I don't need to think too much about the APIs.*". 5 participants who tried INTENT before TF-Coder explicitly mentioned that they wished they could have the dataflow visualization when using TF-Coder. P17 said "*I cannot see the data flow, modify the generated code, provide constraints compared with the first version, which is not convenient. And I need to search the API online to make sure my result is correct.*"

Second, 10 out of 18 participants marked certain operators as desired or undesired to guide the synthesizer towards the final solution. We observed that such feedback to the synthesizer significantly reduced the synthesis time since it helped prune the search space of the synthesizer. As P17 said, "*the ability to add constraints makes me help synthesize the program quickly and flexibly.*" By contrast, when using TF-Coder, participants can only change the input-output examples to disambiguate their intent. In particular, 6 TF-Coder users stopped tuning their examples in the middle of the task and tried to solve the task by themselves, searching for solutions online or coming up with their own solutions.

Third, INTENT supports in situ program editing and validation, so users do not have to switch to an external test environment. When using TF-Coder, 8 participants opened a Jupyter notebook in Google Colab and manually edited and tested a synthesized program. By contrast, no participants switched to Google Colab or any IDE when using INTENT. P9 said, "*[INTENT] felt more of a one-stop shop that helped me accomplish the task without having to use external stuff like trying out code locally myself or searching documentation online.*." P14 said, "*I can test my transformation in real-time which saves me a lot of time in testing.*" In the post-task

	Task 1 - Easy		Task 2 - Medium		Task 3 - Hard	
	Control	Experiment	Control	Experiment	Control	Experiment
	7	10	12	10	12	6
	8	4	8	4	12	4.5
	6	10	11	4	Incomplete	5.5
	12	3	Incomplete	6	10	4
	8	1.75	Incomplete	7.5	11.5	7.5
	2.5	5	7.5	7	12	4.5
Average Time	**7.25**	**5.63**	**13.08**	**6.41**	**12.92**	**5.33**
Overall average time for all tasks combined					**11.08**	**5.79**

Table 1: Task completion time. Incomplete means the participant did not finish the assigned task within 20 min. Completion time colored in grey means the participant provided a wrong solution to the assigned programming task.

survey, 4 participants mentioned that they spent a lot of effort validating the synthesized code when using TF-Coder. P5 said, "*[I need] a convenient way to evaluate the synthesized program.*" P2 said, "*it would be great to have an integrated Python/TF session so that I can directly try the result in the real programming environment.*"

6.2 User Confidence and Cognitive Overhead

In the post-task survey, participants self-reported their confidence in the synthesized code in a 7-point Likert scale question (1 means not confident at all and 7 means highly confident). As shown in Figure 8, participants reported higher confidence when using INTENT, 6.72 vs. 5.44 specifically. This mean difference of 1.28 is statistically significant (unpaired t-test: t = -3.16, df = 20, p-value = 0.005). This confidence improvement was largely attributed to a better understanding of synthesized code when using INTENT. P1 said, "*the dataflow diagram provided a better guarantee of the correctness of the result.*" P2 said, "*the dataflow graph helps me better understand the synthesized expression and gives me higher confidence about the result.*" P15 said, "*[*INTENT*] really helps me understand what the APIs are doing with the input, teach me what they do or confirm my existing knowledge at its worst.*"

Since INTENT has more sophisticated features than TF-Coder, one may wonder if INTENT leads to more mental demand. After each task, participants self-reported the cognitive load of using the assigned tool in a NASA TLX questionnaire. As shown in Figure 9, participants felt less mental demand, effort, and stress when using INTENT. The reasons are two-fold. First, INTENT provides an intuitive dataflow visualization of a synthesized program, which renders individual steps and intermediate results. Thus, participants did not have to refer to external resources, such as TensorFlow API documentation, to understand the synthesized code. Second, with the quick validation feature in INTENT, participants no longer needed to switch to an external test environment to validate the correctness of synthesized code. P10 said, "*when working with complex expressions, it is indeed helpful to understand which function is contributing to which part of the computation. It feels very intuitive to validate the synthesized expression [with* INTENT*], which would have been overwhelming otherwise.*"

6.3 User Ratings of Individual Features

In the post-task survey, participants rated the key features of INTENT. Figure 10 shows the distribution of user ratings. We found

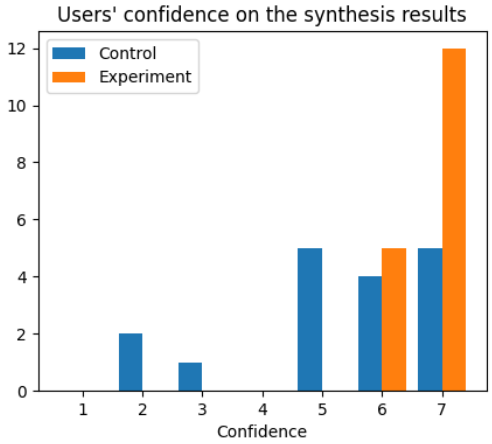

Figure 8: The distribution of participants' confidence on the synthesis results when using INTENT **vs. TF-Coder**

that the majority of participants were satisfied with each feature in INTENT. Among all features, the dataflow visualization is the most appreciated feature. 14 of 18 participants strongly agreed that "*it was helpful to see the dataflow diagram of synthesized code.*" Furthermore, 13 participants agreed or strongly agreed that "*it was helpful to see the element-wise data provenance.*" Marking an operator as desired or undesired was the second most useful feature. 15 out of 18 participants agreed or strongly agreed that "*it was helpful to mark transformation operators as desired or undesired in the data flow diagram.*" P7 said, "*[I like* INTENT *better] because we can add/drop operations if I wanted to optimize the expression generated by the synthesizer.*" In addition, 12 participants agreed or strongly agreed that "*it was helpful to directly modify a synthesized program and get the instant feedback on test results.*" P14 said, "*I can test my transformation in real-time which saves me a lot time in testing.*" P17 said, "*sometimes the generated code is just a little bit incorrect, it would be very helpful if I can modify it and then test it.*"

6.4 User Preference and Feedback

All participants reported that they preferred to use INTENT when writing TensorFlow code. We coded participants' responses to the question about what they like about INTENT. We identified three

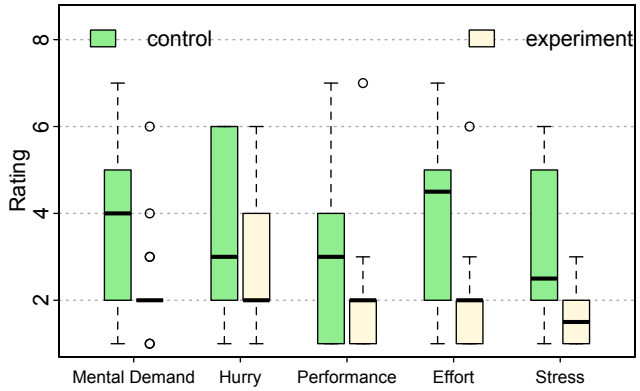

Figure 9: NASA Task Load Index Ratings. Entries with a star mean a statistically significant difference between the control and experiment groups.

themes. First, 17 participants mentioned INTENT helped them understand and validate the program easily. P8 said, "*it allows me to know what exactly happens in each step of operation, thus increasing the system transparency and my understanding of the system.*" P11 said, "*The dataflow graph with descriptions for each function is clear to understand. It makes me find the right solution quickly.*" Second, 5 participants pointed out that INTENT helped them validate the synthesis results. P9 said, "*it helps me build a more accurate mental model to validate the correctness of code.*" Third, 5 participants liked INTENT since it gives them more control over the synthesis process. P4 said, "*giving desirable operations would (hopefully) mean faster inference of a small expression.*"

In the post-task survey, we also asked participants what additional features may help them better solve the task. 3 participants complained about the synthesis speed and hoped to see the progress of the synthesizer. 3 participants mentioned that it was tedious to come up with new examples or corner cases to validate a synthesized program. They wished to have some automated test generation tools to facilitate the validation of synthesized code. Finally, 4 participants suggested it would be helpful to provide a direct link to the official documentation of each TensorFlow operator in the dataflow visualization for their convenience.

7 DISCUSSION

7.1 Design Implications

The user study results suggest that interaction support for program comprehension and validation in the synthesis cycle can significantly improve the productivity of the human-synthesizer team as well as users' confidence on a synthesizer. In the past, most research effort has been put into improving synthesis algorithms or applying program synthesis to new application domains. Our work shows that it is not sufficient to only optimize the synthesis algorithm without considering how users may make use of the synthesized code. Instead, we should treat a human user and a synthesizer as a team and optimize their collaborative performance.

A program synthesizer is essentially an AI agent for programming. Therefore, it suffers from common usability issues of AI agents, e.g., lack of trust, lack of control, overreliance on AI, the interpretability challenge, miscommunication and misinterpretation, etc. [2, 8, 14, 19, 29, 46]. INTENT addresses the lack of trust and lack of control issues by (1) visualizing the intermediate steps in a synthesized program in a dataflow diagram with element-wise data provenance, (2) supporting in situ program editing and validation, and (3) allowing users to mark desired or undesired operators to guide the synthesis process. Based on the user study results, we found that without such interaction support, the task solving process was significantly stagnated.

The dataflow visualization is helpful for programming domains that involve heavy mathematical or sequential computation. For example, SQL query synthesis [47, 50] is another domain where the dataflow visualization can help one understand and validate synthesized code. Specifically, a SQL query can be visualized as a dataflow diagram where intermediate steps are atomic query operations such as filter, join, sort, and intermediate results are SQL tables. By navigating through the dataflow diagram of a SQL query, users can easily understand how the final query result is computed step by step. However, there are some domains where it is more straightforward for users to directly look at the final program output instead of intermediate steps, such as visualization synthesis [48], or where intermediate steps are hard to visualize or comprehend, such as assembly code synthesis [15]. For such domains, dataflow visualization may not be a good fit.

7.2 Handling Multi-Dimensional Tensors

Though the previous examples only demonstrate INTENT on one-dimensional tensors, INTENT supports multi-dimensional tensors as well. In the Supplementary Material, we have included more screenshots to demonstrate how multi-dimensional tensors are rendered in INTENT. Specifically, to ease the specification of multi-dimensional tensors, INTENT visualizes them as *nested tables* in the specification panel (Figure 1 ①). Each dimension in the table is assigned a unique background color to distinguish different dimensions.

If a tensor has too many dimensions (e.g., a 10D tensor), the dataflow visualization can become cluttered; yet, this may not be a major concern in practice, as real-world tensor transformation tasks often do not involve tensors with too many dimensions. To confirm this, we manually checked the TF-Coder benchmark, which includes 70 tensor transformation tasks collected from an internal Google forum and StackOverflow. Among these 70 tasks, none of them involve tensors with more than 4 dimensions. Specifically, 2 of them involve the transformation of 4D tensors, 9 involve 3D tensors, and the rest only involve 1D or 2D tensors.

Since some tensor operators make use of all input elements for all output elements, this might lead to many red dotted lines when rendering data provenance. Among the 101 tensor operators supported by INTENT, we only find 13 such operators, such as `tf.reduce_sum` and `tf.broadcast_to`. In the Supplementary Material, we have included screenshots of the resulting visualizations involving those operators. We found these visualizations were not cluttered since the tensors have four dimensions or less. Furthermore, since the data provenance is only rendered on demand when users click an

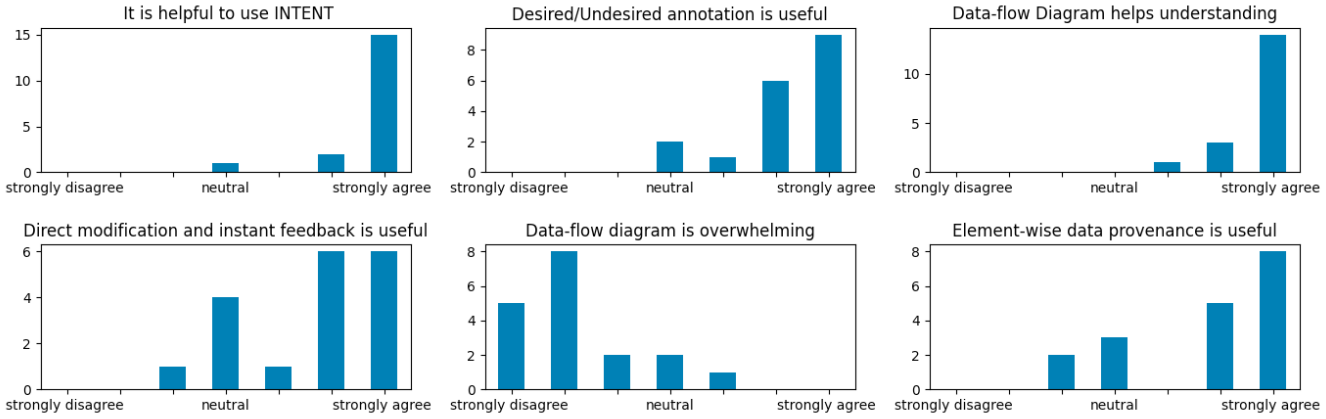

Figure 10: Users' ratings on features

element in a tensor, those data provenance lines are unlikely to intersect with each other.

7.3 Target User Groups and Use Cases

INTENT is designed to empower users who already have programming experience but are unfamiliar with deep learning. Note that programmers who are experienced in one domain (e.g., web development) can still struggle with DL due to the cryptic APIs in TensorFlow and lack of conceptual and math background in DL, as shown in prior work [3]. Furthermore, several participants in our user study who were experts in TensorFlow also appreciated INTENT, since they did not have to recall which APIs to use during implementation.

INTENT is not designed for end-users without programming experience. Therefore, our design and findings do not generalize to end-user programming. Indeed, it is hard for end-users to read a dataflow diagram or directly edit a program. To support end-user programming, one should consider converting synthesized programs to a representation that is comprehensible to end-users. For example, Mayer et al. [31] proposed translating the synthesized code to English descriptions to help end-users understand its behavior. Zhang et al. [54] proposed to generate distinguishing examples and corner cases to help end-users validate the synthesized code.

7.4 Limitations and Future Directions

Our current user study design limits us to necessarily attributing participants' success to individual features in INTENT. One can improve the study design by capturing the utility rate of each feature or comparing with variants with individual features disabled. In the post-task survey, participants rated the usefulness of each feature (Figure 10). Though this is a subjective measurement, it to some extent indicates which feature is more helpful. Among all features, the dataflow visualization is the most appreciated feature.

INTENT normally takes more than 30 seconds or even longer to synthesize a program, which may stand in the way of rapid, iterative specification of a program. This challenge is not unique to INTENT, but rather a byproduct of working with program synthesizers with a large search space. We believe distributed computing can be leveraged to speed up program synthesis significantly. The search process in program synthesis is highly parallelizable. Furthermore, it may also be helpful to allow a synthesizer to quickly return some imperfect but promising solutions, so users can provide some early feedback, which would enable more rapid iteration to incrementally reach a final, perfect solution.

Currently, INTENT can only generate TensorFlow code that uses tensor transformation APIs. As a future direction, it would be appealing if INTENT could be extended to support the construction of an entire neural network. This would require supporting more TensorFlow APIs in the synthesis framework, such as `tf.keras.layers.LSTM`. In addition, input-output examples may not be descriptive enough to specify the desired model. Instead, users may find it more intuitive to communicate model specifications in natural language (e.g., "*the model should have an attention layer*") or constraints (e.g., `!GPU && Memory >= 16GB`). Other modalities such as sketching is also worth considering to support rich specification modalities in neural network synthesis.

8 CONCLUSION

This paper presents INTENT, an interactive program synthesizer that generates TensorFlow code to transform tensors on behalf of users. Since tensor transformation often involves multi-dimensional arrays and sophisticated linear algebra operations, INTENT provides an interactive dataflow visualization to render intermediate results and element-wise data provenance in a tensor transformation program. With this feature, users can conveniently track how the final tensor and individual elements in it are computed step by step. INTENT also supports in situ program editing and validation to help users establish trust on the synthesis results and explore alternative solutions. A user study with 18 programmers shows that participants finished assigned tasks with only half the time, higher success rate, and more confidence when using INTENT compared to using a baseline synthesizer without the aforementioned features.

ACKNOWLEDGMENTS

We would like to thank anonymous participants for the user study and anonymous reviewers for their valuable feedback. We would also like to thank Yitao Huang and Lyubing Qiang for their contributions to an early prototype of INTENT.

REFERENCES

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International conference on computer aided verification*. Springer, 934–950.

[2] Gagan Bansal, Tongshuang Wu, Joyce Zhou, Raymond Fok, Besmira Nushi, Ece Kamar, Marco Tulio Ribeiro, and Daniel Weld. 2021. Does the whole exceed its parts? the effect of ai explanations on complementary team performance. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.

[3] Carrie J Cai and Philip J Guo. 2019. Software developers learning machine learning: Motivations, hurdles, and desires. In *2019 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, 25–34.

[4] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

[5] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 487–502.

[6] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. *IBM AI helps to break down massive code to ease cloud migration*. Retrieved Accessed: 2022-03-29 from https://www.ibm.com/blogs/research/2021/02/ai-refactoring-cloud-migration/

[7] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–12.

[8] Malin Eiband, Sarah Theres Völkel, Daniel Buschek, Sophia Cook, and Heinrich Hussmann. 2019. When people and algorithms meet: User-reported problems in intelligent everyday applications. In *Proceedings of the 24th international conference on intelligent user interfaces*. 96–106.

[9] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 614–626.

[10] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 317–330.

[11] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99.

[12] P. Guo. 2018. *How Did People Write Machine Learning Code in the Past?* Retrieved 2022-03-29 from https://cacm.acm.org/blogs/blog-cacm/230805-how-did-people-write-machine-learning-code-in-the-past/fulltext

[13] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[14] Kevin Anthony Hoff and Masooda Bashir. 2015. Trust in automation: Integrating empirical evidence on factors that influence trust. *Human factors* 57, 3 (2015), 407–434.

[15] Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 134–148.

[16] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1219–1231.

[17] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11)*. Association for Computing Machinery, New York, NY, USA, 3363–3372. https://doi.org/10.1145/1978942.1979444

[18] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[19] Rafal Kocielnik, Saleema Amershi, and Paul N Bennett. 2019. Will you accept an imperfect ai? exploring designs for adjusting end-user expectations of ai systems. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–14.

[20] David Kurlander, Allen Cypher, and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.

[21] J. Landauer and M. Hirakawa. 1995. Visual AWK: a model for text processing by demonstration. In *Proceedings of Symposium on Visual Languages*. 267–274. https://doi.org/10.1109/VL.1995.520818

[22] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (Oct. 2009), 65. https://doi.org/10.1609/aimag.v30i4.2262

[23] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1719–1728.

[24] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6038–6049.

[25] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. Appinite: A multi-modal interface for specifying data descriptions in programming by demonstration using natural language instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.

[26] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*. 577–589.

[27] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

[28] Greg Little, Tessa A Lau, Allen Cypher, James Lin, Eben M Haber, and Eser Kandogan. 2007. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 943–946.

[29] Ewa Luger and Abigail Sellen. 2016. " Like Having a Really Bad PA" The Gulf between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 5286–5297.

[30] Mehdi Manshadi, Daniel Gildea, and James Allen. 2013. Integrating Programming by Example and Natural Language Programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence* (Bellevue, Washington) *(AAAI'13)*. AAAI Press, 661–667.

[31] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *28th ACM User Interface Software and Technology Symposium (UIST 2015)* (28th acm user interface software and technology symposium (uist 2015) ed.). ACM – Association for Computing Machinery. https://www.microsoft.com/en-us/research/publication/user-interaction-models-for-disambiguation-in-programming-by-example/

[32] Richard G McDaniel and Brad A Myers. 1999. Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 442–449.

[33] Robert Miller and Brad Myers. 2002. LAPIS: smart editing with text structure. (01 2002). https://doi.org/10.1145/506443.506447

[34] Francesmary Modugno and Brad A. Myers. 1997. Visual Programming in a Visual Shell-A Unified Approach. *J. Vis. Lang. Comput.* 8 (1997), 491–522.

[35] Brad A Myers and William Buxton. 1986. Creating highly-interactive and graphical user interfaces by demonstration. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 249–258.

[36] Brad A Myers, Jade Goldstein, and Matthew A Goldberg. 1994. Creating charts by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 106–111.

[37] Brad A. Myers and Richard McDaniel. 2001. Chapter 3 - Demonstrational Interfaces: Sometimes You Need a Little Intelligence, Sometimes You Need a Lot. In *Your Wish is My Command*, Henry Lieberman (Ed.). Morgan Kaufmann, San Francisco, 45–III. https://doi.org/10.1016/B978-155860688-3/50004-X

[38] Brad A Myers and Richard McDaniel. 2001. Demonstrational interfaces: sometimes you need a little intelligence, sometimes you need a lot. In *Your Wish is My Command*. Elsevier, 45–III.

[39] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1114–1124.

[40] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Dan Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal Program Inference: a Marriage of Pre-trained Language Models and Component-based Synthesis. In *OOPSLA*. https://www.microsoft.com/en-us/research/publication/multi-modal-program-inference-a-marriage-of-pre-trained-language-models-and-component-based-synthesis/

[41] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *IJCAI 2015* (ijcai 2015 ed.).

[42] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian

Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
[43] Christopher Scaffidi, Brad Myers, and Mary Shaw. 2008. Topes. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 1–10.
[44] Kensen Shi, David Bieber, and Rishabh Singh. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. *CoRR* abs/2003.09040 (2020). arXiv:2003.09040 https://arxiv.org/abs/2003.09040
[45] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.
[46] Alan R Wagner, Jason Borenstein, and Ayanna Howard. 2018. Overtrust in the robotic age. *Commun. ACM* 61, 9 (2018), 22–24.
[47] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1631–1634. https://doi.org/10.1145/3035918.3058738
[48] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.
[49] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: filtering spreadsheet data using examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 195–213. https://doi.org/10.1145/2983990.2984030
[50] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.
[51] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A colorful approach to text processing by example. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 495–504.
[52] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.
[53] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–115.
[54] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.
[55] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. 2021. Unveiling the mystery of api evolution in deep learning frameworks a case study of tensorflow 2. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 238–247.

A SYNTHESIS ALGORITHM

Our algorithm is presented in Algorithm 1. It takes as input a set *OP* of tensor operators (e.g., TensorFlow APIs) and aims to generate programs using these operators. Each operator in *OP* is associated with a *cost*, which will later be used to guide the search. The overall goal here is to find a program that satisfies the provided examples with a smallest cost, guided by other types of specifications.

Towards this goal, the algorithm begins by assigning costs to operators based on the user-provided examples E, natural language description N, as well as the desired/undesired operators D and U (lines 1-4). In a nutshell, these costs are constructed using machine learning models. For instance, we use the multilayer perceptron (MLP) model from [44] to update costs based on examples (line 2) and their TF-IDF model to handle natural languages (line 3). Both models are trained in a supervised manner; we refer interested readers to prior work [44] for details of these models.

Different from prior work, our work additionally leverages the operators preference information; that is, we also adjust costs based on the desired operators D and undesired operators U (line 4). Specifically, INTENT divides the cost of a desired operator by 3 whereas it multiplies the cost of an undesired operator by 3. The factor of 3 is designed empirically. Note that we choose to *penalize* undesired operators instead of completely removing them from the pool of candidate operators; this is because user feedback may not always be correct, especially in the domain of tensor transformation where many tasks often involve complex, non-intuitive linear algebra operations. Users may not be aware of some operations that are indeed useful and may mistakenly mark them as undesired. For example, given the task "*setting all values to zeros except for ones*" as well as the following input/output example,

`in:[3, 1, 4, 5, 1, 8]` → `out:[0, 1, 0, 0, 1, 0]`

it is tempting to mark `tf.cast` (a TensorFlow operator that changes an tensor's data type) as undesired since it does not explicitly participate in any numerical value calculation but actually it is a important intermediate step to convert boolean mask to a binary vector, which can be multiplied to the input tensor to clear all ones in the input. The correct program is shown below.

```
tf.multiply(in, tf.cast(tf.equal(in, tf.constant(1)),
                        tf.int32))
```

By only penalizing operators that are mistakenly labeled as undesired, INTENT still has a chance to generate the correct program with these operators after trying many other operators.

Lines 5-16 describe the bottom-up search process. Because the search space grows exponentially in the program size, we perform the so-called *value-based memoization* [1, 45] to optimize the search performance. At a high-level, this optimization clusters programs based on their outputs. That is, if two programs produce the same output, they will be treated as "one single program" effectively as the search algorithm progresses. Since multiple programs may yield the same output value, this optimization is able to reduce the search space quite dramatically (i.e., it searches the *space of program values*, rather than the *space of programs*). More specifically, the algorithm maintains a set V of values that current enumerated programs generated. V is initialized to contain only the user-provided constants and input tensors from the given examples (line 5); however, V grows as search progresses (line 14) by incorporating new values that are produced by newly discovered programs (line 10). Note that, the search is structured by first enumerating the costs T in the ascending order (line 6), which effectively prioritizes programs with smaller costs (i.e., less complex programs) over more complex programs with larger costs. Given T, the algorithm constructs all programs that have cost T (lines 7-16). In particular, it considers all operators in *OP* (line 7), and for each *op*, it executes *op* on current values in V (lines 8-10). Line 11 performs value-based memoization: we add v to V only if v is *not* yet seen in the past. Note that lines 12-13 also update the meta-data associated with v; for example, it records the cost of this program. Finally, at line 14, it returns the current program if it already matches our examples E. This program is also guaranteed to have the smallest cost.

B DATA PROVENANCE COMPUTATION

Formally, we define *element-wise data provenance specification* for a tensor operator *op* as a function that maps from any element in the output tensor to a set of elements in the input tensors. Specifically, consider $Y = f(X_1, \cdots, X_n, c_1, \cdots, c_m)$, where f is a function that transforms n input tensors $X_1, \cdots, X_n$ to an output tensor Y using

Algorithm 1: Tesnfor transformation synthesis algorithm.

Input: A set of input-output tensor examples $E = \{(I_i, O_i)\}$, a natural language description N, a set of tensor operators OP where each operator is associated with a weight, a set of user-provided constants C, a set of desired operators D, a set of undesired operators U

Output: A tensor transformation program P such that $\forall_{(I_i,O_i)\in E}\ P(I_i) = O_i$.

$OP \leftarrow$ InitializeOperatorCosts();
$OP \leftarrow$ UpdateCostsByExample(OP, E);
$OP \leftarrow$ UpdateCostsByNL(OP, N);
$OP \leftarrow$ UpdateCostsByPref(OP, D);
$V \leftarrow$ CreateInitialValues(C, E);
for $T = 1, 2, \cdots$ **do**
  forall $op \in OP$ **do**
    $n \leftarrow op.arity$; $w \leftarrow op.weight$;
    forall $\vec{a} = [a_1, a_2, \cdots, a_n] \in V^n$ *s.t.* $T = w + \sum_i a_i.weight$ and $\vec{a}$ *is valid arguments for* op **do**
      $v \leftarrow$ Execute(op, $\vec{a}$);
      if $v \notin V$ **then**
        $v.weight \leftarrow T$;
        $v.prog \leftarrow op(a_1.prog, \cdots, a_n.prog)$;
        $V \leftarrow V \cup v$;
        if $v.prog$ satisfies E **then**
          return $v.prog$

m additional non-tensor parameters $c_1, \cdots, c_m$. The element-wise data provenance specification S for f would be of the form:

$$S_{(f,Y,X_1,\cdots,X_n,c_1,\cdots,c_m)}(\mathbf{v}) \coloneqq \{(X_i, \mathbf{u}) \mid Y[\mathbf{v}] \text{ is computed from } X_i[\mathbf{u}]\}.$$

Here, $\mathbf{v}$ is a vector (of indices) denoting the position of an element in the output tensor Y, and $\mathbf{u}$ is the position of an element in the input tensor X_i. This equation means computing the element at position $\mathbf{v}$ of Y "uses" the element at position $\mathbf{u}$ in X_i.

Let us illustrate this using the following example. Consider a TensorFlow function `tf.roll(X, shift=s, axis=a)` which shifts the tensor elements on a specified dimension. For instance, given inputs $X = [[1, 2, 3, 4], [8, 6, 4, 2]]$ and $s = 1, axis = 1$, it gives output $Y = [[4, 1, 2, 3], [2, 8, 6, 4]]$. Here, `tf.roll` shifts each element by 1 along axis 1. The following equations describe the provenance relationship in detail for this `tf.roll` operator.

$$S_{(roll,Y,X,1,1)}([0,0]) = \{[0,3]\} \quad S_{(roll,Y,X,1,1)}([1,0]) = \{[1,3]\}$$
$$S_{(roll,Y,X,1,1)}([0,1]) = \{[0,0]\} \quad S_{(roll,Y,X,1,1)}([1,1]) = \{[1,0]\}$$
$$S_{(roll,Y,X,1,1)}([0,2]) = \{[0,1]\} \quad S_{(roll,Y,X,1,1)}([1,2]) = \{[1,1]\}$$
$$S_{(roll,Y,X,1,1)}([0,3]) = \{[0,2]\} \quad S_{(roll,Y,X,1,1)}([1,3]) = \{[1,2]\}$$

More generally, the provenance specification is encoded symbolically in INTENT. For instance, the symbolic provenance specification for `tf.roll` is shown as follows.

$$S_{(roll,Y,X,s,a)}([v_1, \cdots, v_a, \cdots, v_l]) \coloneqq \{(X, [v_1, \cdots, (v_a - s)\%X.shape[a], \cdots, v_l])\}$$

This equation specifies that the v_a-th element on the a-th dimension in Y is computed using the $\big((v_a - s)\%X.shape[a]\big)$-th element on the a-th dimension from X.

To reduce the manual effort of writing data provenance specifications, we used the automatic differentiation of TensorFlow to automatically generate provenance for 59 out of 101 TensorFlow operators that are used in INTENT (Appendix C). We briefly explain how this automated generation process works as follows.

Suppose $f : \mathbb{R}^{m\times n} \to \mathbb{R}^{p\times q}$ is a differentiable TensorFlow function and $X \in \mathbb{R}^{m\times n}$ is a tensor input. First, we obtain the output tensor $f(X)$. Then, if we want to know the provenance of $f(X)[\mathbf{v}]$, we can propagate the target output element $f(X)[\mathbf{v}]$ through the differentiable function f to derive the gradient with respect to X

$$D = \frac{\partial f(X)[\mathbf{v}]}{\partial X} \in \mathbb{R}^{m\times n}$$

If $D[\mathbf{u}] \neq 0$, $f(X)[\mathbf{v}]$ depends on $X[\mathbf{u}]$ and therefore a data provenance is derived.

$$S_{(f)}(\mathbf{v}) = \{\mathbf{u} \mid D[\mathbf{u}] \neq 0\}$$

On the other hand, since most of the remaining operators are discrete (such as `tf.argmax`), we manually write their specifications. This is because automatic differentiation does not always work for discrete operators. To reduce this manual effort, we grouped similar TensorFlow functions into classes and manually wrote the specification for each class. These classes are listed in Appendix D.

C DIFFERENTIABLE TENSORFLOW FUNCTIONS

(*) mark means the function is not differentiable w.r.t this input tensor, which will be handled specially.

```
tf.abs(x)
tf.add(x, y)
tf.boolean_mask(tensor, mask)
tf.broadcast_to(input, shape)
tf.divide(x, y)
tf.exp(x)
tf.expand_dims(input, axis)
tf.gather(params, indices)
tf.gather(params, indices, axis, batch_dims)
tf.math.cumsum(x, axis)
tf.math.cumsum(x, axis, exclusive=True)
tf.math.divide_no_nan(x, y)
tf.math.negative(x)
tf.math.reciprocal(x)
tf.math.reciprocal_no_nan(x)
tf.math.segment_max(data, *segment_ids)
tf.math.segment_mean(data, *segment_ids)
tf.math.segment_min(data, *segment_ids)
tf.math.segment_prod(data, *segment_ids)
tf.math.segment_sum(data, *segment_ids)
tf.math.unsorted_segment_max(
     data, *segment_ids, num_segments)
tf.math.unsorted_segment_mean(
     data, *segment_ids, num_segments)
tf.math.unsorted_segment_min(
     data, *segment_ids, num_segments)
tf.math.unsorted_segment_prod(
     data, *segment_ids, num_segments)
tf.math.unsorted_segment_sum(
```

```
      data, *segment_ids, num_segments)
tf.math.squared_difference(x, y)
tf.matmul(a, b)
tf.maximum(x, y)
tf.minimum(x, y)
tf.multiply(x, y)
tf.pad(tensor, paddings, mode='CONSTANT')
tf.pad(tensor,
     paddings, mode='CONSTANT', constant_values)
tf.pad(tensor, paddings, mode='REFLECT')
tf.pad(tensor, paddings, mode='SYMMETRIC')
tf.reduce_max(input_tensor)
tf.reduce_max(input_tensor, axis)
tf.reduce_mean(input_tensor)
tf.reduce_mean(input_tensor, axis)
tf.reduce_min(input_tensor)
tf.reduce_min(input_tensor, axis)
tf.reduce_prod(input_tensor, axis)
tf.reduce_sum(input_tensor)
tf.reduce_sum(input_tensor, axis)
tf.reshape(tensor, shape)
tf.reverse(tensor, axis)
tf.roll(input, shift, axis)
tf.sort(values, axis)
tf.sort(values, axis, direction='DESCENDING')
tf.sqrt(x)
tf.square(x)
tf.squeeze(input)
tf.squeeze(input, axis)
tf.subtract(x, y)
tf.tensordot(a, b, axes)
tf.tile(input, multiples)
tf.transpose(a)
tf.transpose(a, perm)
tf.where(condition)
tf.where(condition, x, y)
```

D SPECIFICATION CLASSES

BROADCASTABLE

```
tf.equal(x, y)
tf.greater(x, y)
tf.greater_equal(x, y)
tf.not_equal(x, y)
```

STACKLIIKE

```
tf.add_n(inputs)
tf.concat(values, axis)
tf.stack(values, axis)
```

ALONGAXIS

```
tf.argmax(input, axis)
tf.argmin(input, axis)
```

ALONGSEG

```
tf.math.segment_max(*data, segment_ids)
tf.math.segment_mean(*data, segment_ids)
tf.math.segment_min(*data, segment_ids)
tf.math.segment_prod(*data, segment_ids)
tf.math.segment_sum(*data, segment_ids)
tf.math.unsorted_segment_max(
     *data, segment_ids, num_segments)
tf.math.unsorted_segment_mean(
     *data, segment_ids, num_segments)
tf.math.unsorted_segment_min(
     *data, segment_ids, num_segments)
tf.math.unsorted_segment_prod(
     *data, segment_ids, num_segments)
tf.math.unsorted_segment_sum(
     *data, segment_ids, num_segments)
```

ARGSORT

```
tf.argsort(values, axis, stable=True)
tf.argsort(
     values, axis, direction='DESCENDING', stable=True)
```

CONDITION

```
tf.argsort(
     values, axis, direction='DESCENDING', stable=True)
tf.boolean_mask(tensor, mask)
```

ELEMENTWISE

```
tf.cast(x, dtype)
tf.clip_by_value(t, clip_value_min, clip_value_max)
tf.constant(value)
tf.math.ceil(x)
tf.math.floor(x)
tf.round(x)
tf.sign(x)
```

NO-TENSOR-INPUT

```
tf.eye(num_rows)
tf.eye(num_rows, num_columns)
tf.eye(num_rows, dtype)
tf.fill(dims, value)
tf.one_hot(indices, depth)
tf.ones(shape)
tf.ones_like(input)
tf.range(start)
tf.range(start, limit, delta)
tf.zeros(shape)
tf.zeros_like(input)
```

OTHERS

```
tf.gather_nd(params, indices)
tf.gather_nd(params, indices, batch_dims)
tf.math.bincount(arr)
tf.math.count_nonzero(input)
tf.math.count_nonzero(input, axis)
tf.scatter_nd(indices, updates, shape)
```

```
tf.searchsorted(
     sorted_sequence, values, side='left')
tf.searchsorted(
     sorted_sequence, values, side='right')
tf.sequence_mask(lengths)
tf.sequence_mask(lengths, maxlen)
```

```
tf.shape(input)
tf.tensor_scatter_nd_update(
     tensor, indices, updates)
tf.unique_with_counts(x)
tf.unstack(value, axis)
```